A PROLOG-BASED SYSTEM FOR HARDWARE VERIFICATION

THESIS

Kevin L. Sparks
Captain, USAF

AFIT/GCE/ENG/91M-05

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCE/ENG/91M-05

A PROLOG-BASED SYSTEM FOR HARDWARE VERIFICATION

THESIS

Kevin L. Sparks
Captain, USAF

AFIT/GCE/ENG/91M-05

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1991	*Master's Thesis*

4. TITLE AND SUBTITLE

A Prolog-Based System for Hardware Verification

5. FUNDING NUMBERS

6. AUTHOR(S)

Kevin L. Sparks, Capt USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCE/ENG/91M-05

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

With the expanding number of components provided on a single digital chip, verification of digital designs is becoming a major problem. The more circuits one places on a single chip, the greater the number of input/ouput combinations which need to be checked. A paper by Barrow in 1984 discusses a Prolog-based hierarchical formal verification system which he calls VERIFY. Barrow provided a lot of information on what VERIFY can and cannot do, and on projected enhancements. He does not, however, mention how VERIFY actually performs the task of formal verification. This thesis will provide a description of one possible implementation of the formal verification methodology described in VERIFY.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Formal Verification, Hardware Verification, Prolog	125
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

AFIT/GCE/ENG/91M-05

A PROLOG-BASED SYSTEM FOR HARDWARE VERIFICATION

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Kevin L. Sparks, B.S.

Captain, USAF

March, 1991

Approved for public release; distribution unlimited

Table of Contents

List of Figures

List of Tables

AFIT/GCE/ENG/91M-05

Abstract

With the expanding number of components provided on a single digital chip, verification of digital designs is becoming a major problem. The more circuits one places on a single chip, the greater the number of input/ouput combinations which need to be checked. A paper by Barrow in 1984 discusses a Prolog-based hierarchical formal verification system which he calls VERIFY. Barrow provides information on what VERIFY can and cannot do, and on projected enhancements. He does not, however, mention how VERIFY actually performs the task of formal verification. This thesis will provide a description of one possible implementation of the formal verification methodology described in VERIFY.

A PROLOG-BASED SYSTEM FOR HARDWARE VERIFICATION

I. Introduction

1.1 Background

In any design environment, the designer would like to know if his design really works as expected. Did he build the product right? With the expanding number of components provided on a single digital chip, verification of digital designs is becoming a major problem. The more circuits one places on a single chip, the greater the number of input/ouput combinations which need to be checked.

In the Air Force Instititute of Technology (AFIT) School of Engineering, there is a group of students whose educational emphasis is on the design of these Very Large Scale Integrated (VLSI) circuits. These students would like to have better methods of ensuring that a design is correct before it is sent to another agency for fabrication. A paper by Barrow (1)in 1984 discusses a Prolog-based verification system which he calls VERIFY. Barrow provides information on what VERIFY can and cannot do, and on projected enhancements. He did not, however, mention how VERIFY actually performs the task of formal verification. This thesis will provide a description of one possible implementation of the concepts in VERIFY. Once a Prolog verification methodology is provided, the resulting system could be integrated into a larger hardware design system for use by the AFIT VLSI group. A VLSI design system containing verification tools will speed up the design-fabrication cycle and ultimately result in both time and cost savings to AFIT.

1.2 Problem

In this thesis, hierarchical formal verification methods will be applied to the problem of verification of digital circuit designs. Formal verification methods attempt to prove that a circuit's structural description implies a specified behavioral description. A circuit's

structural description is obtained through some synthesis process. At some lower level (i.e., gate, transistor) the behavior and structure are equivalent, forming the basis for deriving behavior at higher levels. A high-level behavioral description can thus be inferred from a circuit's structure. The approach in this thesis will be to derive a behavioral description by combining previously verified modules hierarchically. Barrow's methodology requires each module to be primitive or composed of other modules. A *primitive* module is a low-level module where behavior and structure are the same. The module is described only in a black-box sense of inputs, outputs, and specified behavior. Since a primitive module's behavior and structure are equivalent, structure implies behavior, and all primitive modules are verified. All other modules are composed of primitive modules and other higher level modules. The non-primitive modules must fully specify their structure and expected high-level behavior. All non-primitive modules are verified by first verifying the module's components. If all components are successfully verified, then the behaviors derived during the component's verifications can be substituted for those components while deriving the module's behavior. Assuming this behavioral derivation process produces a valid behavioral description, this derived behavioral description is now compared to the module's specified behavioral description. Barrow elects to show that the derived behavior and specified behavior are equivalent. If the behaviors are equivalent, it is also true that structure, the derived behavior, implies behavior, the specified behavior.

This thesis advocates a specific method of verification of digital hardware design. To understand the specific problem, a general understanding of digital design verification techniques is required.

1.3 Summary of Current Knowledge

Approaches to Verification. The process of ensuring a digital circuit conforms to its specification can be approached in three different ways: formal synthesis, simulation, and formal verification. *Formal synthesis* is a design methodology which ensures that the resulting implementation performs the high-level specified behavior. Such a process would take a high-level behavioral description and automatically generate the proper implementation. Formal synthesis currently remains a lofty and unattainable goal. *Simulation* tries

to ensure that the design meets the expected behavior, also known as the functional or behavioral specification, by applying an appropriate number of inputs and "verifying" that they produce the expected outputs. Most "verification" efforts to date use some form of simulation and do not fully verify a circuit's behavior. Maruyama contends that the problem with simulation is that it "does not definitely ensure the conformance of design to specifications" (12:22). This is the reason formal synthesis or formal verification must be considered. *Formal verification* uses logic to verify a circuit. Each circuit to be verified will consist of some high-level behavioral specification and some technology-dependent physical implementation. Formal verification verifies a circuit by showing that the physical implementation, or structure, implies the high-level specification, or behavior. In logic, it establishes that

$$Structure \Rightarrow Behavior.$$

There are two basic ways to prove that this implication is valid. The first method is to use predicate logic to perform a step-by-step formal proof of this implication. The second method, which is used by Barrow, is to logically determine that the behavior and structure are equivalent.

$$\text{If } Structure \Leftrightarrow Behavior \text{ then, by definition,}$$
$$Structure \Rightarrow Behavior \quad \text{and} \quad Behavior \Rightarrow Structure.$$

The systems used to represent behavior and structure and the methods used to prove that, directly (implication) or indirectly (equivalence), structure implies behavior, provide the major differences between formal verification methodologies. It should be noted that Barrow's VERIFY could be set up to prove only that structure implies behavior; however, Barrow chose to use Prolog to implement his methodology. Prolog's operation is based on pattern-matching, and pattern-matching can, and is, exploited to make equivalence determination much easier than implication proving. The equivalence determination process will be explained in great detail in the Program Development chapter.

Formal Verification Methodologies. The majority of formal verification research appears to be coming from the United Kingdom, and, in particular, from Cambridge

University. Gordon (8:2) feels that for formal verification to be possible for real systems, a verification system must provide both a high-level mathematical formalism for writing specifications, and tools for mechanizing the proof of correctness. Gordon admits that VERIFY contains these necessary qualities, but he advocates the use of predicate logic to both specify and verify circuits. Gordon uses a predicate logic system called Higher-Order Logic, or HOL.

Gordon has provided a defense of the use of HOL for formal verification (9). In that discussion, examples are provided which show how HOL can be used to specify a circuit's black-box behavior, or specification, by imposing constraints on the inputs and outputs. Then, given one possible physical implementation of the device, the physical implementation can be represented logically by another group of constraints on the inputs, outputs, and internal connections. The specifications and implementations provided by a user are expressed as a function which is composed of constraints specified as normal predicate logic statements consisting of quantifiers, propositions, and predicates. This representation can be classified as the "definitional method" as presented by Clocksin (4:63–64), except predicate logic constructs are used as opposed to Prolog syntax. "To verify that the implementation correctly implements the specification, it must be proved that if the inputs and outputs satisfy the constraints imposed by the implementation, then they also satisfy the specification" (9:163). Notice that the implementation must only imply the specification and not be strictly equivalent. This may allow the implementation actually to provide more functionality than the specification. HOL actually performs a formal proof of the implication. This implication is proven in HOL through exhaustive simulation for all inputs (this would show equivalence) in simple cases, by expansion of library component definitions, and even by mathematical induction. The process is comparable to a hand-written formal proof. HOL also provides capabilities for dealing with time (9:169). Barrow's VERIFY provides no methods for dealing with time or performing mathematical induction (1:488–489).

Barrow's modules allow circuits to be verified hierarchically by verifying lower level modules and keeping them in a library for use by larger systems. Using this structural hierarchy to aid in design verification is essential to achieve verification of circuits which

contain more than basic elements. Hwang (11) also uses hierarchy to verify the correctness of finite-state machines. Hwang considers a module to consist of an implementation part and a specification part. The specification is a description of the expected behavior. The implementation can be further divided into the following categories:

1. Interior node module - a module composed of submodules, each containing a specification and implementation.
2. Leaf node module - a set of net lists of elementary gates and latches obtained by some synthesis process.
3. Primitive module - a module which need not be verified, so it can be assumed to satisfy the specification. (11:410)

Hwang argues that most leaf nodes can be modelled as finite state machines and presents a method to verify designs hierarchically. He shows that a finite-state machine representation of the implementation satisfies the finite-state machine representation of the specification if and only if every input sequence applied to a specific state in both the specification and implementation machines results in the same output sequence in both the specification and implementation machines (11:411).

The methodologies of Gordon, and Hwang are viable alternatives, but Barrow's VERIFY methodology will be explored in this thesis. HOL provides greater flexibility in logic specification, time representation, and proof mechanisms, but requires a learning curve which precludes a normal AFIT thesis cycle. Hwang's methodology is very similar to Barrow's, but provided no guidance on a possible implementation. Barrow's VERIFY is implemented in Prolog, which is readily available at AFIT, and is the current required language for the introductory Artificial Intelligence course. The apparent replication of Barrow's VERIFY by Brezocnik et al. (3) and Grabowiecki et al. (10) and the information provided by these efforts added to the viability of a system such as VERIFY.

One should be cautioned that Barrow's VERIFY has limits. The most critical limitation is the lack of a method for temporal representation. Even given this limitation many types of circuits can be verified, and, in the process of verifying those circuits, the methods to provide temporal representation can be studied. This thesis provides a Prolog

implementation of Barrow's formal verification methodology, and no time will be devoted to the determination of a possible method for temporal representaion.

1.4 Approach

The AFIT VLSI group currently "verifies" module corrrectness through simulation. As mentioned above, simulation cannot truly verify module correctness. Some method for formal verification is required. Prolog enables a circuit to be described behaviorally, and also provides predicate-logic constructs for formal verification. Also, Prolog can be developed in a PC environment, which allows rapid prototyping. Barrow has a nice methodology, but will it really work in practice? The need for a formal verification system, Barrow's methodology, and the easy access to Prolog all played a role in the approach utilized in this thesis.

The goal of this thesis was to use Prolog to develop a prototype of Barrow's verification methodology, remembering that the ultimate goal would be to provide a design/verification system to the AFIT VLSI group. The approach was to build the prototype by first verifying a simple combinational logic circuit and then move to the real problem of verifying a sequential circuit. An Exclusive-OR gate was first verified using a Boolean Expansion algorithm created by CPT Dukes (6:39–43). Next Barrow's counter-circuit example was attempted. This attempt proved successful and provided the basic prototype structure to be used for further research. At that point, only modules composed of primitive submodules could be verified. To have a true hierarchical verification system, one would need to provide verification to any depth. It was here that Barrow's exploitation of hierarchy was discovered. Barrow fully verified each submodule in the process of verifying a module. This required every module to be fully specified both behaviorally and structurally. Once a submodule was verified, the submodule's derived behavior (and/or state) could be substituted for the submodule's output (and/or next state) when deriving a module's behavior. This would make all previously verified submodules appear to be primitive, since a simple substitution would be made. A full-adder module containing three levels of structure was verified to show this kind of hierarchical verification. The full-adder verification required the use of a mainframe version of Quintus Prolog due to

the small memory model of PROLOG-1. A PC version of Quintus Prolog will be used in future research.

Once it was shown that a sequential circuit and a module consisting of more than two structural levels could be formally verified, the work on this thesis turned to an explanation of just how this was accomplished. AFIT now has a working prototype to build a library of formally verified circuit modules.

1.5 Organization of Thesis

In the chapters to follow, the prototype developed to perform formal verification will be presented. In the rest of this thesis, the prototype will be identified as AFIT_VERIFY. An introduction to Prolog concepts is included to demonstrate the types of constructs required to use and further develop AFIT_VERIFY. The formal verification methodology of Barrow and its implementation, AFIT_VERIFY, are explained in detail. This discussion also includes an example of a module description which is used to verify a circuit. Next, the actual development of AFIT_VERIFY is discussed. The body of the thesis concludes with a summary of the results and recommendations for further development. Appendix A provides a listing of all code developed for AFIT_VERIFY, and Appendix B contains some sample verification runs using PROLOG-1 and Quintus Prolog. The sample runs provide a flavor of the type of processing required to verify even small circuits. The runs also contrast the capabilities of PROLOG-1 and Quintus Prolog.

II. Introduction to Prolog

2.1 Introduction

To achieve a full understanding of the subsequent chapters, which discuss the implemented code, one must have some basic understanding of Prolog and how it works. This chapter will explain how Prolog differs from traditional languages such as PASCAL, it will explain how a Prolog program is created, and it will discuss the capabilities of Prolog which make it appropriate for formal verification.

Artificial Intelligence programmers should be well-versed in both Lisp and Prolog (2:vii–ix). Languages like PASCAL are how-type or procedural programming languages. The programmer must specify *how* a specific procedure is to be accomplished. Some believe that Lisp is the "champion" of the how-type languages. Prolog, on the other hand, is a what-type or declarative language. In essence, one states logically what needs to be done and Prolog does it. At least it appears that way. A note of caution is due here. "Plentiful experience and devotion to conventional procedural programming, for example in PASCAL, might be an impediment to the fresh way of thinking Prolog requires"(2:xii).

Prolog, which stands for PROgramming in LOGic, is rooted in mathematical logic. In particular, the syntax of Prolog is first-order predicate logic written in Horn clause form (2:61). A Horn clause is a logical implication with no more than a single atom as consequent. For example:

$$A \Leftarrow B_1 \& \ldots \& B_n$$

In this case, an atom is meant to be some indivisible concept. The level of indivisibility depends on the level of abstraction used to describe a particular problem.

Clocksin and Mellish (5:240–244) created a Prolog program to convert a first-order predicate calculus formula into clause form. The clause form produced by Clocksin and Mellish is not Horn clause form, but a mechanical process exists to convert the Clocksin and Mellish clause form into Horn clause form. This process requires redefining an atom as the negation of another atom. Then the redefined atom can be moved across the implication. For example, given the following clause form,

$$A \vee B \Leftarrow D\&E,$$

and the fact that

$$C \Leftrightarrow \tilde{}B,$$

the resulting Horn clause would be the following:

$$A \Leftarrow C\&D\&E.$$

Although this mechanical process is nice to have, it is generally not used. Most problem statements can be manipulated and placed in Horn clause form. Using the Horn clause form allows Prolog to mechanically prove user-supplied theorems, by using the resolution principle introduced by Robinson (13).

A complete understanding of mathematical logic or the theory behind resolution is not required to create some very powerful programs using Prolog. The basic mechanisms of Prolog are pattern matching, tree-based data structuring, and automatic backtracking. Prolog is known as a goal-directed language. This is "because Prolog accepts questions as goals that are to be satisfied" (2:7). This will be made much clearer in the examples to follow.

2.2 Prolog Syntax

Prolog has only a single data object, the *term*. A term is either a *constant*, a *variable*, or a *structure*. A *constant* can be either a number or an atom. An *atom* is any sequence of characters, except control characters, which begins with a lower case letter. (Atoms beginning with an upper case letter must be enclosed in single quotes.) A *variable* must begin with an upper case letter or the underline character. The underline character, _, when used alone, is known as the anonymous variable. The anonymous variable is used when its value is irrelevant. It can be thought of as a "don't care" value. Examples of atoms and variables appear in Table 2.1

"A *structure* is composed of an atom, called the *principal functor*, followed by a sequence of terms called *components* of the structure" (7:2-6). Structures are stored as

Table 2.1. Prolog atoms vs. variables

Atoms	Variables
a	A
an_Atom	Var
'an_Atom'	A_variable
'Prolog-1'	VAR1

trees with the principal functor being the root and the components as the offspring. Any component which is also a structure appears as a subtree under the principal functor. For example, the structure `triangle(point(1,1),A,point(2,3))`, which represents a family of triangles containing the points (1,1),(2,3), and some unknown point, A, would be stored as shown in Figure 2.1.

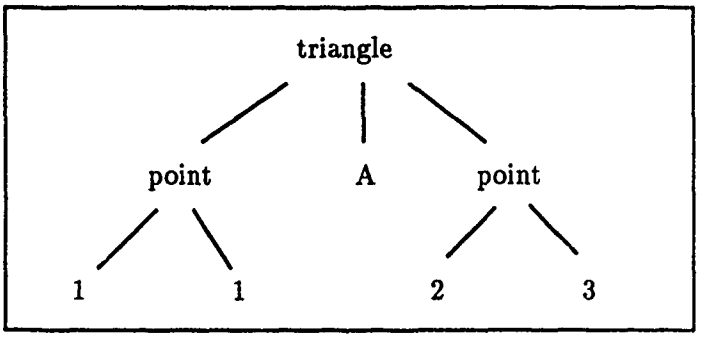

Figure 2.1. Triangle structure as seen by Prolog.

The following structure,

```
parent(tom,bob),
```

consists of the principal functor **parent** and the two components `tom` and `bob` which, in this case, happen to be atoms. The above **parent** structure will be used to explain how one can use Prolog in a declarative manner to solve problems.

The family relation problem is a classic introductory problem in Artificial Intelligence and especially in Prolog. The structure `parent(tom,bob)` means Tom is the parent of Bob and, in Prolog, is called a *fact.* As with other programming languages the atoms `parent`, `tom`, and `bob` mean nothing in particular to Prolog; the programmer provides the semantics.

Prolog merely recognizes the pattern **parent(tom,bob).** One can assert many such facts to represent the family tree in Figure 2.2.

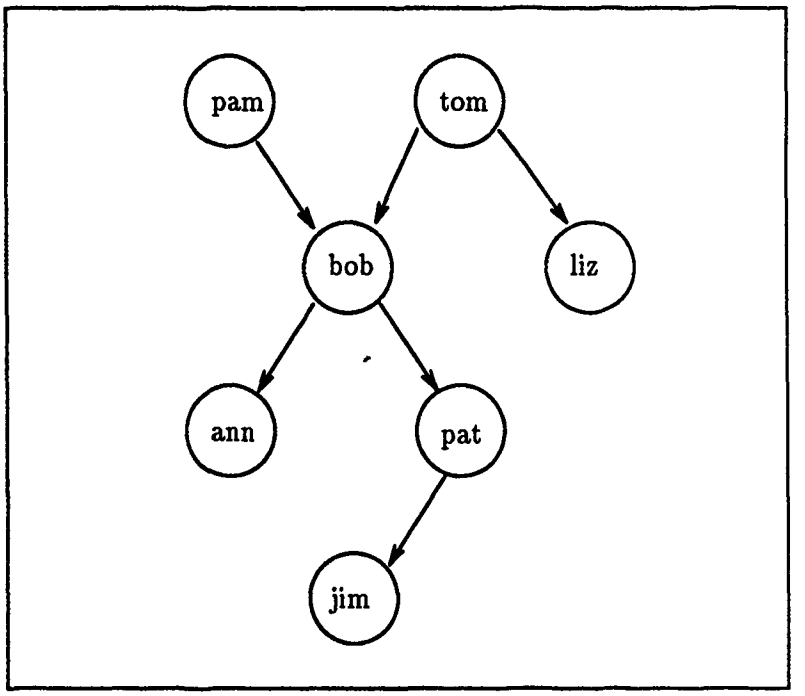

Figure 2.2. Sample Family Tree.

```
parent(pam,bob).
parent(tom,bob).
parent(tom,liz).
parent(bob,ann).
parent(bob,pat).
parent(pat,jim).
```

These facts would constitute a very simple Prolog program consisting of six clauses. After loading the program one could ask "Is Bob the parent of Pat?" by typing the question

```
?- parent(bob,pat).
```
(?- is the Prolog prompt)

Prolog would find this asserted fact and answer **yes.** One could also ask "Who is the parent of Bob?" by typing the question

```
?- parent(X,bob).
```

Prolog would then answer

```
X = pam,
```

and, if Prolog is told to provide more answers ("y" in Prolog-1 and ";" in Quintus Prolog), it would reply

```
X = tom,
```

and, if more answers are requested, Prolog would reply

```
no.
```

Granted, this may not seem useful to a non-database programmer. One might think that if he has to provide the program with all the facts to begin with, then he really doesn't need the program. It should be noted that Prolog can be viewed as a relational database, and the programs written are just queries to that database. This view of a program would be foreign to most functional programmers. AFIT_VERIFY was developed using Prolog's declarative programming style. The resulting code shows more resemblance to a standard language than to a database query language due to Prolog's added deductive capabilities and flexible query mechanisms.

Prolog *clauses* come in three varieties: *facts*, *rules*, and *questions*. A Prolog clause consists of the *head* and the *body*. The head and body are separated by the :- operator. For example:

```
offspring(Y,X) :-          head
 parent(X,Y).              body
```

The :- operator represents the implication arrow, $\Leftarrow$. So , the Prolog clause really means that if the body is true, then the head is true. The body is a list of goals separated by commas (logical AND) or semicolons (logical OR), and the head provides the pattern to be searched for by Prolog. A *fact* is something that is always true, which is represented

by a clause with an empty body. A *question* only has the body, which is represented by a clause with an empty head. A *rule* declares that something (the head) is true provided a given condition (the body) is true (2:13). This can be shown by Table 2.2 and the following example.

Table 2.2. Prolog clauses.

Form	Type
$H \Leftarrow$	fact
$\Leftarrow T$	question
$H \Leftarrow T$	rule

Given the six previously defined **parent** facts, one could now create a grandparent rule. A grandparent is a parent of a parent. In Prolog, all variables are assumed to be universally quantified, so logically stated,

for all X and Y,

```
X is a grandparent of Y if
    X is a parent of Z and
    Z is a parent of Y.
```

This is easily stated in Prolog as:

```
grandparent(X,Y) :-
     parent(X,Z),
     parent(Z,Y).
```

Now, if one asks the question **grandparent(X,pat)**, Prolog will respond with

```
X = pam,
```

and then, if more answers are required,

```
X = tom.
```

One can see that even though no grandparent facts exist, using the grandparent rule one can derive all grandparent-grandchild relationships by using the question

```
?- grandparent(X,Y).
```

The previous example is carried out in much more detail in Bratko (2). Other rules about families illustrate some subtle problems. A sister is a female with the same parents as another sibling. This logic results in a girl's being her own sister. Prolog looks as if it does so much (and it does) that a programmer can begin to believe that Prolog is thinking and forget to think for himself. Prolog simply goes through the database searching for possible instantiations of variables to solve a user-provided goal. The search can be speeded up by reducing the search space through the introduction of *cuts*, which allow a goal to be satisfied only once with the same instantiation. This is when programming in Prolog gets dangerous, because some solutions have been eliminated from the search space. The programmer is now telling Prolog *how* to get a solution as opposed to *what* solution is needed. As much as possible, one should use the strengths of Prolog and its what-type capabilities.

Another major feature of Prolog, which is necessary to tackle problems of any significance, is *recursion.* A simple example of this is the predecessor relation (2:15). A parent is a predecessor. So is a grandparent, great-grandparent, great-great-grandparent, ... A `predecessor` clause could be defined as,

```
predecessor(X,Y) :-
      parent(X,Y).
predecessor(X,Y) :-
      parent(X,Z),
      parent(Z,Y).
predecessor(X,Y) :-
      parent(X,Z),
      parent(Z,A),
      parent(A,Y).
        ⋮ ,
```

and **predecessor** clauses could be added to a sufficient depth to cover all cases in the family tree. But isn't a predecessor more simply defined as either a parent or a predecessor of a parent? This is easily written in Prolog as

```
predecessor(X,Y) :-
    parent(X,Y).
predecessor(X,Y) :-
    parent(X,Z),
    predecessor(Z,Y).
```

This new definition will cover predecessors to any depth and demonstrates the power of Prolog to encapsulate a large search problem into a few simple clauses.

It should be noted that multiple clauses with the same principal functor and number of components (arity) perform the logical OR function of the clause bodies. The separate goals of the two clauses could appear in a single **predecessor** clause separated by a semicolon, as follows:

```
predecessor(X,Y) :-
    parent(X,Y)
    ;
    (parent(X,Z),
    predecessor(Z,Y)).
```

However, the use of separate clauses enhances readability and is the recommended Prolog programming practice. Prolog merely searches for the matching clause when trying to satisfy a goal. If the first clause fails, Prolog just moves to the next matching clause. This also brings up the point that ordering of clauses plays a role in the efficiency of the search. The clauses which will be satisfied most frequently in a program should be placed at the top of the database (i.e., should occur first in the program).

2.3 Summary

Although these examples are simple, they do provide the basis for an understanding of Prolog. The most important concepts to grasp are that rules are simply stated as, "This pattern holds if the following goals can be proven". Since the ultimate goal of this research is to verify large circuits hierarchically, extensive use of recursion will be required in the development of the system described in this thesis.

The final point to make about Prolog concerns modification. In the chapters to follow, the code should be explained in sufficient detail to allow modification without requiring a major revision. Generally, to handle a new case, one just adds another clause. This is quite acceptable, and this type of programming lends itself to follow-on efforts. An interested person can take the code developed in this effort and simply add new clauses to cover the types of designs of interest to future research.

This chapter was not intended to provide a thorough tutorial on Prolog, but simply to allow the reader to understand the basic concepts. The use of cuts and ordering of clauses are explained in some detail in the references (2, 7). As with most languages, the best way to get an understanding of Prolog is just to write some programs.

III. VERIFY

3.1 Introduction

The thrust of this thesis is to provide a Prolog implementation of the formal verification methodology described by Barrow (1). Barrow developed a program in Prolog called VERIFY which tries to prove the correctness of digital designs (1:437). The key principle underlying VERIFY is that, "given the behaviors of components of a system and their interconnections, it is possible to derive a description of the whole system" (1:445). His program is based on the assumption that a design is "comprised of modules, organized hierarchically, and modeled as finite state machines"(1:440). The modules are described by stating the inputs, the expected outputs, and any internal state information (1:440). These statements make up a structural description. VERIFY takes the information in the structural description and derives a behavioral description (1:450). Using the inherent pattern matching capabilities of Prolog and some added mathematical techniques, VERIFY then tries to prove the equivalence of the derived behavioral description and the specified behavioral description. The mathematical techniques include methods for algebraic manipulation, simplification, canonicalization (i.e., putting equations in a normal form for easier comparison), evaluation, and case analysis (1:452–454). If the derived behavioral description and the specified behavioral description can be proved to be equivalent, then the module specification is verified; however, there is still the possibility that the original module specification is incorrect. That is a completely different problem which Barrow doesn't attempt to address (1:445).

Brezocnik et al. (3) and Grabowiecki et al. (10) have reported successful replication of Barrow's process. None of these "VERIFY" efforts provided significant detail on how to replicate the verification code. An actual implementaion of the verification methodology described in VERIFY (1) is the product of this thesis. Barrow's methodology is best understood by the presentation of an example. The counter-circuit example given by Barrow (1:440–445) will be discussed using the Prolog clauses as developed for this thesis.

3.2 Module Structure

Given the counter-circuit in Figure 3.1, a Prolog description for the counter is the following:

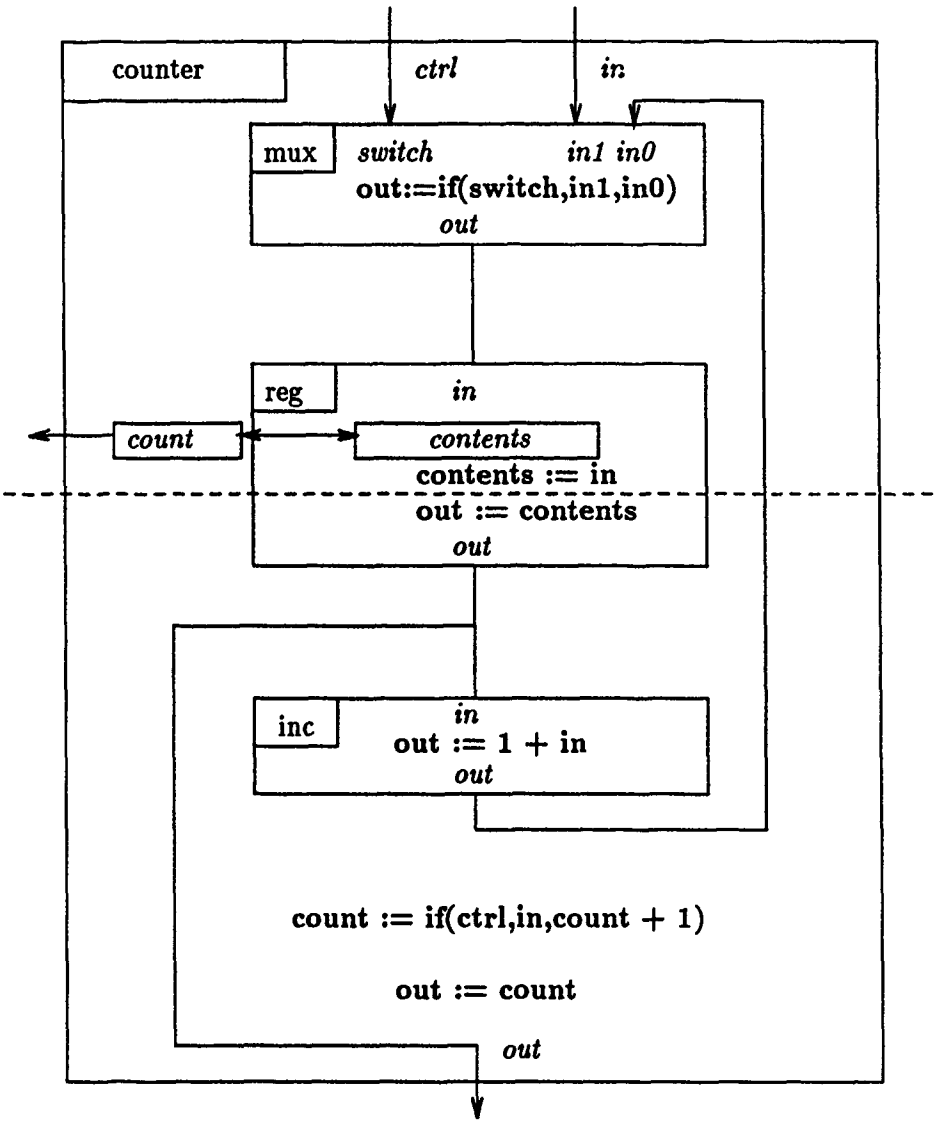

Figure 3.1. Barrow's counter-circuit example.

```
/*****************************************************/
/*                                                   */
/*  Counter.pro                                      */
/*                                                   */
/*     Module definitions for the counter example    */
/*     in Barrow's VERIFY article.                   */
/*                                                   */
/*****************************************************/

/*----------------- INCREMENTER ---------------------*/

module_name(inc).

port(inc,in(AnInc),input,integer).
port(inc,out(AnInc),output,integer).

/*  Behavior Specification */

output_eqn(inc, out(AnInc) := 1 + in(AnInc)).

/*---------------  MULTIPLEXER ---------------------*/

module_name(mux).

port(mux,in0(AMux),input,integer).
port(mux,in1(AMux),input,integer).
port(mux,switch(AMux),input,boole).
port(mux,out(AMux),output,integer).

/*  Behavior Specification */

output_eqn(mux,  out(AMux) :=  if(switch(AMux),
                                  in1(AMux),
                                  in0(AMux))).
```

```
/*--------------------  REGISTER --------------------*/

module_name(reg).

port(reg,in(AReg),input,integer).
port(reg,out(AReg),output,integer).

/*  Behavior Specification */

state_of(reg,contents(AReg),integer).

output_eqn(reg,out(AReg) := contents(AReg)).

state_eqn(reg,contents(AReg) := in(AReg)).

/*--------------------  COUNTER ----------------------*/

module_name(counter).

port(counter,in(ACounter),input,integer).
port(counter,ctrl(ACounter),input,boole).
port(counter,out(ACounter),output,integer).

part(counter,muxA(ACounter),mux).
part(counter,regA(ACounter),reg).
part(counter,incA(ACounter),inc).

connected(counter,ctrl(ACounter),switch(muxA(ACounter))).
connected(counter,in(ACounter),in1(muxA(ACounter))).
connected(counter,out(muxA(ACounter)),in(regA(ACounter))).
connected(counter,out(regA(ACounter)),in(incA(ACounter))).
connected(counter,out(incA(ACounter)),in0(muxA(ACounter))).
connected(counter,out(regA(ACounter)),out(ACounter)).

/*  Behavior Specification */

state_of(counter,count(ACounter),integer).
state_map(counter,count(ACounter),contents(regA(ACounter))).

output_eqn(counter,out(ACounter) := count(ACounter) ).

state_eqn(counter, count(ACounter) :=  if(ctrl(ACounter),
                                          in(ACounter),
                                          count(ACounter) + 1)).
```

The counter-circuit comprises a register (reg), a multiplexer (mux), and an incrementer (inc). Each module must contain its own **module_name** which uniquely identifies each module to be verified or later used as a component for a larger module. Each module has ports which are associated with the specific module. In the **port** clause,

```
port(Module,Name,I/O,Type),
```

Name uniquely identifies each input/output port for the module with module_name, **Module**. **I/O** distinguishes between input/output ports and the signal type (boole,integer,...) is identified by **Type**.

The current definition of a 'primitive' module is one which contains no **part** clauses. The **part** clause,

```
part(Module,Name,SubModule),
```

is the method to hierarchically define the **Module**. The **Name** attribute is the identifier for a specific **SubModule**. **Name** is supplied as **SubModuleA(AModule)**, **SubModuleB(AModule)**, ... **SubModules** are verified prior to deriving the behavior of **Module**. Using Prolog variables (**AModule**) as part of the **Name** specification provides a direct method of instantitiating each **SubModule** behavior as part of the **Module** behavior. This provides for Barrow's substitutional model for deriving behavior. A **Module's Submodule's** derived behavior is merely substituted for the **SubModule's** output at the appropriate time in the process of deriving the **Module's** behavior.

Deriving a **Module's** behavior makes use of the **part**, **connected**, **state_of**, **state_map**, **output_eqn**, and **state_eqn** clauses. The process of deriving behavior and state will be discussed in the Program Development chapter. The **connected** clause,

```
connected(Module,Source,Destination),
```

describes all interconnections of a **Module**. Each module connection is specified from **Source** to **Destination** with **SubModule** identifiers included to provide the substitutional method for deriving behavior previously mentioned.

Each output of a module is specified in an **output_eqn** clause,

```
output_eqn(Module, Output := Specified_Behavior),
```

which provides the **Specified_Behavior** to be compared to the behavior derived from the structural specification (i.e., **port**, **part**, and **connected** clauses).

At this point in the AFIT_VERIFY development, and, as reported by Barrow, only modules with internal feedback loops broken by state variables (typically registers) (1:450–451) have been considered. The dotted line in Figure 3.1 represents this conceptual break. The **state_of**, **state_map**, and **state_eqn** clauses allow state specification. The state_of clause,

```
state_of(Module,State,Type),
```

simply identifies each state variable to the external environment. This allows the specification of **Module** as a black box. The **state_map** clause,

```
state_map(Module,State,Internal_State),
```

shows the actual internal component, **Internal_State**, which contains the actual State. Again, **State** and **Internal_State** contain variables to allow the derivation of the expected next state via the substitutional process. In the **state_eqn** clause,

```
state_eqn(Module,State := Next_State),
```

Next_State is now the specified behavior to progress to the next State. Next_State is compared to a derived next state to determine equivalence.

3.3 AFIT_VERIFY Methodology

AFIT_VERIFY is invoked by typing

```
verify(Module)
```

after loading the appropriate **Module** descriptions and code (see Appendix 2 for sample runs). The **verify** clauses serve as a main program. First, the **verify** clauses recursively verify each **SubModule**. If all **SubModules** are successfully verified, the behavior of each of **Module's** outputs is derived and then compared to the specified behavior for each output. A **derived_behavior(Module,Output,Derived_Behavior)** fact for each equivalent output is asserted into the database for later use as a verified **Derived_Behavior** for the **Module**. If all derived behaviors and specified behaviors are equivalent (a **Module** with no state is verified at this point), all next states are derived and compared to their specified next states. Each equivalent next state is asserted as a **next_state(Module,State,Next_State)** fact into the database for later use as a verified **Next_State** for the **Module**. If this is successful, the module is verified and a **verified(Module)** fact is asserted into the database for future use by other modules. Various issues of what to assert into the database, when to assert, and how to assert will be discussed in the Program Development chapter.

3.4 Summary

Barrow provided a methodology for performing hierarchical formal verification. This included an algorithm and a specific method to provide behavioral and structural specifications. The algorithm: verify a module's components; derive the behavior of all outputs/states from the module's structure; and determine if each derived output/state is equivalent to its specified output/state, is followed by AFIT_VERIFY. Barrow's specification syntax is also strictly adhered to when possible. This specification syntax provides Prolog with specific facts about a module. AFIT_VERIFY then performs one possible implementation of Barrow's algorithm on these module facts.

IV. Program Development

4.1 Introduction

Based on Barrow's work (1), AFIT_VERIFY was developed in three major portions. The first section recursively verifies a module's components. The next section derives a behavioral description from the provided structural description, and the final section determines the equivalence of the derived and specified behavioral descriptions. Before explaining what each portion of code provides, some comments on the development environment and methodology will be disussed.

4.2 Development Environment

The majority of code was developed on a PC using PROLOG-1. This caused some minor modifications to Barrow's naming conventions since the clause, `state`, has special meaning to PROLOG-1. The clause **`state_of`** is used in place of Barrow's `state` clause. The PC environment allowed for rapid prototyping by using Sidekick to make coding changes and then popping back into PROLOG-1 to run the improved code.

PROLOG-1 has some shortcomings, the most significant being insufficient stack space. This was realized during the verification of Barrow's counter-circuit example (1.440–445). During the process of determining the equivalence of the sum output's specified and derived behaviors, the stack space was exceeded. This presented two alternatives: make the Boolean expansion code (6) more efficient, or move to a larger system. Work continued on the AFIT Microvax system Cub using Quintus Prolog since it became apparent PROLOG-1 would not be sufficient for larger modules.

The counter-circuit example was verified on Cub, but not without some minor portability problems. Quintus Prolog has no **`not`** clause defined. (In Prolog, `not` is understood to mean that no clause of the specified type exists. The goal, `not parent(X,Y)` succeeds when no **`parent`** clauses appear in the database.) More importantly, the operator precedences in Quintus Prolog run from 0 to 1200 but only from 0 to 255 in PROLOG-1. Also, the clause **`module`**, has system significance to Quintus; therefore, `module` was changed to **`module_name`**. Finally, Quintus Prolog provides better error-checking mechanisms. One

of these mechanisms checks for the redefinition of a clause when a new file is loaded. A programmer cannot redefine clauses in separate .pro files unless a `multifile` directive is included for each clause which needs to be redefined. Unless the AFIT_VERIFY code is to be kept as one huge .pro file, redefinition of clauses in separate files would be required, since each new module would require redefinition of the `module_name`, `part`, `port`, ...clauses. The `multifile` directives are included in the qops.pro file. Two separate operator files, qops.pro (Quintus) and ops.pro (PROLOG-1), were created to provide a mechanism for system portability. Additional operator files can be created for use on other Prolog systems. In fact, follow-on work is expected to be completed on a PC version of Quintus Prolog, and discrepencies between the PC and mainframe versions should be handled in the operator files.

4.3 Verify

The first goal was to verify a simple combinational circuit, like an Exclusive-OR gate comprising NAND gates. This proved to be simple. The next step was to verify a sequential circuit, one which contained a single state variable. The example demonstrated by Barrow and implemented in AFIT_VERIFY was a simple counter-circuit.

This task was not trivial and introduced many new concepts to the design. After studying a similar verification attempt by Brezocnik et al. (3), it finally became apparent that a modular and easily expandable design should be used. This required a relook at the code used to verify the combinational circuit.

The first attempt at a verification procedure looked like this:

```
verify(Module) :-
   verify_components(Module),
   derive_behavior(Module,out(X),Derived_Behavior),
   equal_behaviors(Module,Derived_Behavior),
   asserta(verified(Module)).
```

In the spirit of declarative programming, Prolog was told to perform the three parts of Barrow's VERIFY algorithm: verify the components, extract a behavior from the struc-

ture, and then compare the extracted behavior to the specified behavior. This approach works well for a combinational circuit with a single output, but what about a sequential circuit or circuits with multiple outputs? Since Prolog is, in essence, a relational database, it was decided to use the database capabilities to provide and update the appropriate state information. This is a variation of the technique used by Grabowiecki et al. (10:40). All parts of the program would have access to the asserted facts. It was also determined that the **verify** clauses would operate as drivers. In so doing, the **verify** clauses would be required to handle the different types of modules to be verified. This required a different **verify** clause for each type of module (i.e. previously verified, primitive, no state, ...). This also created the need for helper procedures to generate and verify multiple outputs, states, and components.

The different types of modules to be verified are handled by the following verify clauses:

```
verify(Module) :-                          /* previously verified module   */
   verified(Module),
   !,
   writeln(['>>>',Module,' previously verified >>>']).
verify(Module) :-                        /* primitive module with no state  */
   not part(Module,_,_),
   not state_eqn(Module,_),
   !,
   asserta(verified(Module)),
   writeln(['>>>',Module,' primitive (needs no verification)>>>']).
verify(Module) :-                          /* primitive module  with state  */
   not part(Module,_,_),
   !,
   asserta(verified(Module)),
   writeln(['>>>',Module,' primitive (needs no verification)>>>']).
verify(Module) :-                          /* non-primitive with no state  */
   not state_eqn(Module,_),
   writeln(['>>> Attempting to verify ',Module,'>>>']),
   verify_components(Module),
   !,
   derive_and_equate_behaviors(Module),
   asserta(verified(Module)),
   writeln(['<<< Success! Behavior of',Module,'meets its specification.']).
```

```
verify(Module) :-                          /* non-primitive with state  */
   writeln(['>>> Attempting to verify ',Module,'>>>']),
   verify_components(Module),
   !,
   derive_and_equate_behaviors(Module),
   derive_and_equate_states(Module),
   asserta(verified(Module)),
   writeln(['<<< Success! Behavior of',Module,'meets its specification.']).
```

These five clauses should handle any type of **Module** to be verified. Some decisions were made about primitives which would impact other portions of the code. For a primitive module, behavior is equivalent to structure, or more importantly, the derived behavior is the specified behavior. At this time, no **derived_behavior(Module,Output,Behavior)** or **next_state(Module,State,Nextstate)** facts are asserted into the database for primitive modules since this information can be obtained directly from the output_eqn(Module, Output := Behavior) and **state_eqn(Module, State := Nextstate)** clauses. If it is later decided that the **derived_behavior** and **next_state** clauses should be asserted, then a clause can be removed from the **derive_behavior** clauses (Rule 2A). A similar decision is required for the **asserta(verified(Module))** for a primitive **Module**. With only one possible **verified** clause per **Module**, the space required seems minimal for the time savings. The time savings can be achieved due to the use of cuts (!) in the **verify** clauses. The cuts remove the need for resatisfaction of goals like **part(Module,_,_)** as Prolog works its way through the clauses.

The helper clauses **derive_and_equate_behaviors**, **derive_and_equate_states**, and **verify_components** allow the **verify** clauses to hierarchically verify each component of a **Module** and ensure that all outputs and states are equivalent. Since the **fail** clause always returns false, the second clauses in the helper procedures check that the first clause did the appropriate work. The second clauses in **derive_and_equate_behaviors** and **derive_and_equate_states** count the number of derived and specified behaviors. If the numbers are equal, then the first clause derived and equated all the outputs/states. This assumes that the procedures determining the equivalence of the derived and specified behaviors are correct. If these clauses fail, then cleanup can be performed by retracting the asserted clauses. The **verify_components** clauses create a list of the module's com-

ponents (**parts**) using **set_of** and checks that a **verified** fact has been asserted using **parts_verified**. It should be noted that Quintus Prolog has a system-defined clause **setof(Objects,Predicate,List)** which returns the **List** of the set of **Objects**, which satisfy the **Predicate**. The behavior of **setof** was not as expected. Since PROLOG-1 has no **setof** clause defined, the clauses **set_of**, **find_all**, and **length_of** are provided in both operator files to ensure portability. The helper procedures are as follows:

```
derive_and_equate_behaviors(Module) :-
  derive_behaviors(Module,Output,Derived_Beh),
  equal_behaviors(Module,Output,Derived_Beh),
  asserta(derived_behavior(Module,Output,Derived_Beh)),
  fail.
derive_and_equate_behaviors(Module) :-
  set_of(Outputs,output_eqn(Module,Outputs := _),Outlist),
  length(Outlist,Outnum),
  set_of(Outputs,derived_behavior(Module,Outputs,_),Derlist),
  length(Derlist,Dernum),
  Outnum =:= Dernum.
derive_and_equate_behaviors(Module) :-
  retract(derived_behavior(Module,_,_)),
  fail.

derive_and_equate_states(Module) :-
  derive_states(Module,State,Next_State),
  equal_states(Module,State,Next_State),
  asserta(next_state(Module,State,Next_State)),
  fail.
derive_and_equate_states(Module) :-
  set_of(States,state_eqn(Module,States := _),Statelist),
  length(Statelist,Statenum),
  set_of(States,next_state(Module,States,_),Derlist),
  length(Derlist,Dernum),
  Statenum =:= Dernum.
derive_and_equate_states(Module) :-
  retract(next_state(Module,_,_)),
  fail.

verify_components(Module) :-
  part(Module,_,Component),
  verify(Component),
  fail.
```

```
verify_components(Module) :-
  set_of(Component,part(Module,_,Component),Complist),
  p rts_verified(Complist),
  writeln(['component list is ', Complist]).

parts_verified([]).
parts_verified([Component|Tail]) :-
  verified(Component),
  parts_verified(Tail).
```

This portion of the code should require no foreseeable modifications when AFIT_VERIFY is expanded.

4.4 Deriving Behavior

"One key principle underling VERIFY is that, given the behaviors of components of a system and their interconnections, it is possible to derive a description of the behavior of the whole system" (1:445). The code provided in this section is one possible implementation of VERIFY's method for deriving behavior.

Since some devices have an associated state and some devices are simply combinational logic, different types of behavior must be derived. However, all devices are described by at most two types of equations: output equations and next state equations. Barrow's notation is used in the following representations for a specified behavior:

```
output_eqn(Module, Out := Function)
state_eqn(Module, Next_State := Function).
```

The **state_eqn** notation became very confusing, since the **state_eqn** really referred to the equation for the next state in a sequential machine. To somewhat remove this ambiguity, when a behavior is derived, the following clauses are asserted into the database:

```
derived_behavior(Module,Output,Behavior)
  next_state(Module,State,Function).
```

This helps the programmer to remember that the prototype is really deriving the expected output and next state. Asserting these clauses in the database also frees the

programmer from passing parameters in the heads of clauses. Asserting these clauses is a variation of the technique used by Grabowiecki et al. (10:40).

Now, all that would be required to derive the behavior of a module would be to type:

derive_and_equate_behaviors(Module) and/or
derive_and_equate_states(Module).

These clauses, when invoked by a **verify** clause for the appropriate module types, extract the **Outputs** and **Next_States** from the module specifications and derive the appropriate behavior. The clause, **derive_behaviors** is used to extract the information for outputs and then invoke the **derive_behavior** clause to actually derive the behavior for the specified output. If a **Module** has state, some internal variables may need to be removed from the derived behavior. This is the reason for two **derive_behavior** clauses and the need for the **substitute_state** clause. The code is as follows:

```
derive_behaviors(Module,Form,Behavior) :-              /* no state */
  not state_eqn(Module,_),
  !,
  output_eqn(Module,Form :=Spec_Behavior),
  derive_behavior(Module,Form,Behavior).
derive_behaviors(Module,Form,Behavior) :-        /* has state (!), */
  output_eqn(Module,Form := Spec_Behavior),
  derive_behavior(Module,Form,TBehavior),
  substitute_state(Module,TBehavior,Behavior).   /* might require  */
                        /* the removal of some internal variables. */
```

The majority of work is performed by the **derive_behavior** clauses. These clauses work their way through the **connected** clauses for each output until a module terminal is reached. The process is a simple substitutional one which would be used by humans. For example, given the following NAND and XOR module specifications,

```
/*-----  Structural Specification for 2-input nand ----*/

module_name(nand2).

port(nand2,in0(ANand2),input,boole).
port(nand2,in1(ANand2),input,boole).
port(nand2,out(ANand2),output,boole).

/* Behavioral Specification   */

output_eqn(nand2,
          out(ANand2) := or( neg(in0(ANand2)), neg(in1(ANand2))) ).


/* Structural specification for a two-input Exclusive-OR  */

module_name(xor).

port(xor,in0(AnXor),input,boole).
port(xor,in1(AnXor),input,boole).
port(xor,out(AnXor),output,boole).

part(xor,g1(AnXor),nand2).
part(xor,g2(AnXor),nand2).
part(xor,g3(AnXor),nand2).
part(xor,g4(AnXor),nand2).

connected(xor,in0(AnXor),in0(g1(AnXor))).
connected(xor,in1(AnXor),in1(g1(AnXor))).
connected(xor,in0(AnXor),in0(g2(AnXor))).
connected(xor,out(g1(AnXor)),in1(g2(AnXor))).
connected(xor,out(g1(AnXor)),in0(g3(AnXor))).
connected(xor,in1(AnXor),in1(g3(AnXor))).
connected(xor,out(g2(AnXor)),in0(g4(AnXor))).
connected(xor,out(g3(AnXor)),in1(g4(AnXor))).
connected(xor,out(g4(AnXor)),out(AnXor)).

/* Behavioral Specification for a two-input XOR */

output_eqn(xor,
          out(AnXor) :=  or( and( neg(in0(AnXor)),
                                  in1(AnXor) ),
                             and( in0(AnXor),
                                  neg(in1(AnXor)) ))).,
```

the following output behavior is derived:

```
or(and(in0(_755),
       or(neg(in0(_755)),neg(in1(_755)))),
   and(or(neg(in0(_755)),neg(in1(_755))),
       in1(_755)))
```

The form is canonicalized, but one can see that the outer OR is g4, the ANDs inside the OR are g2 and g3 and the `or(neg(in0(_755)),neg(in1(_755)))` is the output of g1. All gate outputs have simply been replaced with their derived behavior (Since a NAND gate is considered primitive, the derived behavior of a NAND gate is the specified behavior). A human would obtain this by working his way back through the NAND gates, substituting the appropriate output equation for the NAND gates until the terminal inputs are reached. This is a simple recursive process which is duplicated by the `derive_behavior` clauses. Along the way, a simple substitution of a submodule's derived behavior is made when a submodule's output is encountered. Boolean or mathematical equations are further expanded and canonicalized. Any type of behavior not captured by some Boolean or mathematical rule is simply returned unchanged.

The **`derive_behavior`** clauses are arranged in three groups. The first group traverses through the connected statements (Rules 1A and 1B). The next group replaces outputs with their appropriate behaviors (Rules 2A and 2B). The rules in the remaining groups simplify and canonicalize the behavior obtained from the first two groups. All `derive_behavior` clauses have the following form:

```
derive_behavior(Device,Formula,Behavior)
```

`Device` is the gate or component whose behavior is being derived, such as NAND, XOR, or `counter`. `Formula` is the type of behavior to be derived such as `out(X)` or some state variable equivalent such as `in(regA(X))`. `Behavior` is, of course, the resulting derived behavior. The types of clauses developed are the following:

```
derive_behavior(Module, Form, Source) :-
  connected(Module, Source, Form),
  primary_source(Source),
  !,writeln(['Applying Rule 1A to ',Form]).
derive_behavior(Module, Form, Behavior) :-
  connected(Module, Source, Form),
  derived_source(Source),
  !,writeln(['Applying Rule 1B to ', Form]),
  derive_behavior(Module, Source, Behavior).


derive_behavior(Module, Form, Behavior) :-
  Form \== 1,
  Form =.. [F,G],
  part(Module, G, Component),
  not part(Component,_,_),        /* Component is a primitive module */
  output_eqn(Component, Form := OutForm),
  !,writeln(['Applying Rule 2A to ', Form]),
  writeln([Component,'''s output equation:']),
  writeln(['   ', Form, ' := ', OutForm]),
  derive_behavior(Module, OutForm, Behavior).      /* replace gate */
                                   /* inputs with module variables  */
derive_behavior(Module, Form, Behavior) :-
  Form \== 1,
  Form =.. [F,G],
  part(Module, G, Component), /* from cut, no* primitive component */
  /* previously verified due to verify_components in verify clause */
  derived_behavior(Component,Form,OutForm),
  !,writeln(['Applying Rule 2B to ', Form]),
  writeln([Component,'''s derived behavior:']),
  writeln(['   ', Form, ' := ', OutForm]),
  derive_behavior(Module,OutForm,Behavior).       /* replace gate */
                                  /* inputs with module variables  */


derive_behavior(Module, neg(Form), Behavior) :-
  !,writeln(['Applying Rule 3 to ',neg(Form)]),
  derive_behavior(Module, Form, Beh1),
  evaluate1(neg(Beh1), Behavior).
derive_behavior(Module, and(Form1,Form2), Beh) :-
  !,writeln(['Applying Rule 4 to ',and(Form1,Form2)]),
  derive_behavior(Module, Form1, Beh1),
  derive_behavior(Module, Form2, Beh2),
  evaluate1(and(Beh1,Beh2), Beh).
```

```
derive_behavior(Module, or(Form1,Form2), Beh) :-
  !,writeln(['Applying Rule 5 to ',or(Form1,Form2)]),
  derive_behavior(Module, Form1, Beh1),
  derive_behavior(Module, Form2, Beh2),
  evaluate1(or(Beh1,Beh2), Beh).
derive_behavior(Module, if(Cond,Texp,Fexp), Beh) :-
  !,writeln(['Applying Rule 6 to ',if(Cond,Texp,Fexp)]),
  derive_behavior(Module, Cond, NCond),
  derive_behavior(Module, Texp, NTexp),
  derive_behavior(Module, Fexp, NFexp),
  evaluate1(if(NCond,NTexp,NFexp), Beh).
derive_behavior(Module, First + Second, Beh) :-
  !,writeln(['Applying Rule 7 to ',First + Second]),
  derive_behavior(Module, First, Beh1),
  derive_behavior(Module, Second, Beh2),
  evaluate1(Beh1 + Beh2, Beh).
derive_behavior(Module, Form, Form) :-            /* default Rule */
  writeln(['Applying default Rule to ',Form]).
```

The clauses **primary_source** and **derived_source** are introduced in the first two derive_behavior clauses. The **primary_source** and **derived_source** clauses distinguish between a **Module** input (primary_source) and a **Component** input/output (derived_source).

```
primary_source(Source) :-
  Source =.. [_,Arg],
  var(Arg).

derived_source(Source) :-
  Source =.. [_,Arg],
  Arg =.. [_,Arg2],
  var(Arg2).
```

Any new type of behavior needed to expand AFIT_VERIFY would require new derive_behavior clauses. Again, if the decision is made to assert a derived_behavior fact for primitive modules, then Rule 2A could be removed. The evaluate1 procedure is introduced in this code. It is merely a preliminary simplification of the derived behavior. The clause **evaluate1** is simply a test clause which invokes **evaluate_brown** and prints the behavior before and after simplification. The **evaluate_brown** clauses actually perform the simplification and rudimentary canonicalization. As with the derived_behavior

clauses, **evaluate_brown** clauses will need to be added when new behaviors are added. The first three **evaluate_brown** clauses provide basis cases for a behavioral structure, namely a variable, atom, or an elementary structure. The next three clauses provide a method of simplification and canonicalization of the Boolean functions AND, OR, and NOT. Another canonical form may prove quicker; in that case, these clauses would need to be modified accordingly. The identifier **neg** has been chosen as the negation functor, as opposed to the more widely used **not**, since Prolog defines the goal **not** P to succeed in case the goal P fails (i.e., cannot be proved) in the existing database. The clause **not** is not provided in Quintus Prolog. This is why **not** is defined in the qops.pro file. The remaining clauses provide a method of simplification and canonicalization of the operators required to verify the fulladder and counter, namely, **if** and **+**.

```
evaluate1(X,EX) :-
  writeln(['Value of ',X,':']),
  evaluate_brown(X,EX),
  writeln(['          ',EX]).

evaluate_brown(X,X) :-
  var(X),!.
evaluate_brown(X,X) :-
  atomic(X),!.
evaluate_brown(Struct,Struct) :-
  Struct =.. [F,Arg],
  ( var(Arg) ; atom(Arg) ),!.

evaluate_brown(and(X,Y),Value) :-
  evaluate_brown(X,EX),
  evaluate_brown(Y,EY),
  (  (EX  = 0 ;  EY = 0),  !,
     Value = 0
  ;
     EX = 1, !,
     Value = EY
  ;
     EY = 1, !,
     Value = EX
  ;
     !,Value = and(EX,EY)
  ).
```

```
evaluate_brown(or(X,Y),Value) :-
  evaluate_brown(X,EX),
  evaluate_brown(Y,EY),
  (  (EX = 1 ;  EY = 1), !,
      Value = 1
  ;
      EX = 0, !,
      Value = EY
  ;
      EY = 0,  !,
      Value = EX
  ;
      !,Value = or(EX,EY)
  ).
evaluate_brown(neg(X),Value)  :-
  evaluate_brown(X,EX),
  (  var(EX), !,
     Value = neg(X)
  ;
     EX = 0,  !,
     Value = 1
  ;
     EX = 1, !,
     Value = 0
  ;
     atom(EX), !,
     Value = neg(EX)
  ;
     EX = neg(N), !,
     Value = N
  ;
     EX = and(A1,A2),  !,
     evaluate_brown(neg(A1),NA1),
     evaluate_brown(neg(A2),NA2),
     Value = or(NA1,NA2)
  ;
     EX = or(O1,O2), !,
     evaluate_brown(neg(O1),NO1),
     evaluate_brown(neg(O2),NO2),
     Value = and(NO1,NO2)
  ;
     !,Value = neg(EX)
  ).
```

```
evaluate_brown(if(Cond,Texp,Fexp),Value) :-
  evaluate_brown(Cond,NCond),
  evaluate_brown(Texp,NTexp),
  evaluate_brown(Fexp,NFexp),
  (  ( NCond = 1, !,                          /* Condition is true */
       Value = NTexp ) ;                      /*  return True exp  */
     ( NCond = 0, !,                          /* If False then     */
       Value = NFexp ) ;                      /*  return False exp */
     ( NTexp = NFexp, !,                    /* Condition irrelevant */
       Value = NTexp ) ;                      /*  if choices equal */
     Value = if(NCond,NTexp,NFexp),            /* otherwise return */
     !  ).                                  /* simplified expression. */

evaluate_brown(X+Y,Z) :-
  integer(X),
  integer(Y), !,
  Z is X + Y.                 /* force simplification of 1 + 2 = 3 */

evaluate_brown(X+Y,Z) :-
  integer(Y),!,                          /* X not integer due to cut */
  evaluate_brown(X,NewX),
   Z = Y + NewX.                /* canonicalize with integer first */

evaluate_brown(X+Y,Z) :-
  !,evaluate_brown(X,NewX),
  evaluate_brown(Y,NewY),
  Z = NewX + NewY.

evaluate_brown(X,X).          /*  default simplification for complex */
                              /*  structures like in(incA(X)).       */
```

Deriving the next state is only a variant of deriving the behavior of an output. The state variable and its internal variable equivalents are extracted. A behavior is then derived for the internal variable, which is, in fact, the next-state behavior for the state variable. Occurrences of internal variables in the derived behaviors are replaced with the state variables when such simplifications will make the process of equating behaviors simpler (3:103). The `replace_all` clauses find all signals in a derived behavior which are connected to a state variable in a module and replace them with the state variable. First, the `New` signal is substituted for the `Old` signal in the Old Behavior (`OldBeh`) to create an intermediate Substituted Behavior (`SB`). Next, any signal (`Other`) connected to the `Old`

signal is found and the substitution process is repeated (recursive call of replace_all). The code required to derive the next state is the following:

```
derive_states(Module,State,Next_State) :-
  state_of(Module,State,Type),                    /*  It has state  */
  state_map(Module,State,Internal),/* It's mapped to an internal part*/
  state_eqn(Part,Internal := NextState),     /* Internal state is a  */
  derive_behavior(Module,NextState,Beh),   /* function of inputs and */
  substitute_state(Module,Beh,Next_State).         /* previous state */


substitute_state(Module,DerBeh,SubBeh) :-
  state_map(Module,External,Internal),
  !,
  writeln(['Derived Behavior:  ',DerBeh]),
  replaceall(Module,Internal,External,DerBeh,SubBeh),
  writeln(['Substituted Behavior:  ',SubBeh]).

replaceall(Module,Old,New,OldBeh,SubBeh) :-
  replace(Old,New,OldBeh,SB),
  (   connected(Module,Old,Other) ;
      output_eqn(Part,Other := Old) ),
  !,
  replaceall(Module,Other,New,SB,NewSB),
  evaluate1(NewSB,SubBeh).          /* Simplify further if possible */
replaceall(Module,Old,New,Beh,Beh) :- !.      /* no mote connections */

replace(Old,New,Other,Other) :-
  atomic(Other),
  write('Rule2'),nl,!.
replace(Old,New,Other,Other) :-
  var(Other),
  write('Rule3'),nl,!.
replace(Old,New,Other,Other) :-
  Old =.. [F,Arg1],                   /* keeps in(X) = in(incA(X))  */
  Other =.. [G,Arg2],                 /* from occuring, occurs test */
  F \== G,
  ( var(Arg2) ; atomic(Arg2) ),              /* already simplified */
  write('Rule4'),nl,!.
  replace(Old,New,and(X,Y),and(NewB1,NewB2)) :-
  !,write('Rule and'),nl,
  replace(Old,New,X,NewB1),
  replace(Old,New,Y,NewB2).
```

```
replace(Old,New,or(X,Y),or(NewB1,NewB2)) :-
  !,write('Rule or'),nl,
  replace(Old,New,X,NewB1),
  replace(Old,New,Y,NewB2).
  replace(Old,New,neg(X),neg(NewB)) :-
  !,write('Rule neg'),nl,
  replace(Old,New,X,NewB).
replace(Old,New,X + Y,NewB1 + NewB2) :-
  !,write('Rule + '),nl,
  replace(Old,New,X,NewB1),
  replace(Old,New,Y,NewB2).
replace(Old,New,if(Cond,Texp,Fexp),if(NewB1,NewB2,NewB3)) :-
  !,write('Rule if'),nl,
  replace(Old,New,Cond,NewB1),
  replace(Old,New,Texp,NewB2),
  replace(Old,New,Fexp,NewB3).
replace(Old,New,Other,NewB) :-              /* in(X) /== in(incA(X)) */
  Old =.. [F,Arg1],
  Other =.. [F,Arg2],
  ( ( var(Arg1), not var(Arg2) );
    ( not var(Arg1), var(Arg2) ) ),
  replace(Old,New,Arg2,NewArgs),
  NewB =.. [F,NewArgs],          /* Old behavior or Old Behavior is */
  write('Rule struct'),nl,!.      /*  some other nested structure   */
replace(Old,New,Old,New) :-
  write('Rule 1'), nl,!.            /* If you find X replace with Y */
replace(Old,New,Other,Other) :-
   write('Default Rule'),nl.                      /* default rule.  */
```

As with **derive_behavior** and **evaluate_brown**, new **replace** clauses will need to be added when new behavioral structures are created. It should now be readily apparent that Prolog can be thought of as a guided search. A programmer merely provides the type of patterns which he wants to encounter. This is why new behavioral descriptions are accomodated by adding new clauses to "handle" the new structure.

4.5 Determining Equivalence

The majority of work on AFIT_VERIFY went into developing the ability to derive behavior. Determining equivalence of the two behaviors is a well-understood, but non-trivial task. The examples verified have been limited, and the need for extensive work on equivalence will arise with newer and larger examples. Most of the work of determin-

ing equivalence is accomplished by the Boolean expansion code created by CPT Dukes (6:39–43). It is noted that the **Type** attribute in a **port** clause can be used to guide the methods of determining equivalence. The work to this point concentrated on verifying and deriving the behavior of Boolean circuits with the exception of the counter-circuit example. In the counter-circuit example, determining equivalence was simplified by making a simple canonicalization. Any occurrence of (term + integer) was replaced by (integer + term). This allowed the derived behavior and specified behavior of **count(counter)** to be trivially equivalent. The following description and code provides the current methods for determining equivalence.

The **derive_and_equate_behaviors** and **derive_and_equate_states** clauses use the **equal_behaviors** and **equal_states** clauses to provide every output/state equivalence. The primary methods used to determine equivalence are *simplification* and *Boolean expansion.* The following code implements the equivalence portion of AFIT_VERIFY.

```
equal_behaviors(Module,Output,Derived_Beh) :-
   output_eqn(Module,Output := Specified_Beh),/* get specified behavior */
   eqb(Module,Derived_Beh,Specified_Beh).
equal_states(Module,Nextstate,Derived_State):-
   state_eqn(Module,Nextstate := Function),   /* get specified state    */
   eqb(Module,Derived_State,Function).

eqb(M,X,X)  :-                                /* trivial identity       */
   !.
eqb(M,DB,SB) :-                               /*     Boolean expansion  */
/* expandable(M),            fewer than too be determined combinations */
                                             /* and Boolean variables   */
   evaluate_dukes(DB,NewDB),
   evaluate_dukes(SB,NewSB),
   writeln(['Does ',NewDB,' =']),writeln(['      ',NewSB]),
   eq(NewDB,NewSB),
   writeln([DB,' =']),writeln(['     ',SB]),
   writeln(['By Boolean Expansion']),!.

expandable(M) :-                          /* some how_many function will */
   port(M,_,_,boole),!.                   /*  be required                */
```

```
eqb(M,DB,SB) :-                                    /*  simplification */
   evaluate1(DB,NDB),
   evaluate1(SB,NSB),
   ( DB \== NDB ;
     SB \== NSB ),
   writeln(['Derived behavior is: ',DB]),
   eqb(M,NDB,NSB), !.
```

The clause **eq(NewDB,NewSB)** is the driver for the code which performs CPT Dukes's Boolean expansion code. An introduction to the concept of Boolean expansion seems appropriate at this point.

Boole's Expansion. There are two basic methods of determining the equivalence of two functions f and g. Consider a design composed of the following:

$$i = 1 \ldots n \text{ inputs and}$$
$$j = 1 \ldots m \text{ outputs.}$$

Equations f and g are Boolean functions of n input variables with m possible output values. The first method of determining equivalence is to enumerate all possible 2^n input values and compare all m possible output values. This approach will always require $m2^n$ operations and gets out of hand very quickly as m and n grow.

The second approach, which can have many variants, is to perform Boole's Expansion on functions, and show that each expanded sub-function is equivalent. This can cause success much sooner and will probably require much less than 2^n expansions if combined with an effective pattern matching algorithm which checks for equivalence at each node expansion. (This is why Prolog is so beneficial; its entire operation is based on pattern matching.)

Boole's expansion can be stated for n variables as follows (6:36):

Theorem 1: $f(x_1, x_2, \ldots, x_n) \Leftrightarrow x_1' f(0, x_2, \ldots, x_n) + x_1 f(1, x_2, \ldots, x_n)$

A complete expansion of two variables is the following:

$$\begin{aligned} f(x,y) &\Leftrightarrow x' f(0,y) + x f(1,y) \\ &\Leftrightarrow x'(y' f(0,0) + y f(0,1)) + x(y' f(1,0) + y f(1,1)) \end{aligned}$$

This final form is what is known as the Minterm Canonical Form. The equivalence of f and g can be shown (6:36–37) by the following:

Equation 1: $f(x_1, x_2, \ldots, x_n) \Leftrightarrow g(x_1, x_2, \ldots, x_n)$

iff

Equation 2: $(x_1' f(0, x_2, \ldots, x_n) \Leftrightarrow (x_1' g(0, x_2, \ldots, x_n)$

AND

Equation 3: $(x_1 f(1, x_2, \ldots, x_n) \Leftrightarrow (x_1 g(1, x_2, \ldots, x_n)$

Boole's Expansion can then be recursively applied to Equation 2 and Equation 3 to produce the tree in Fig 4.1.

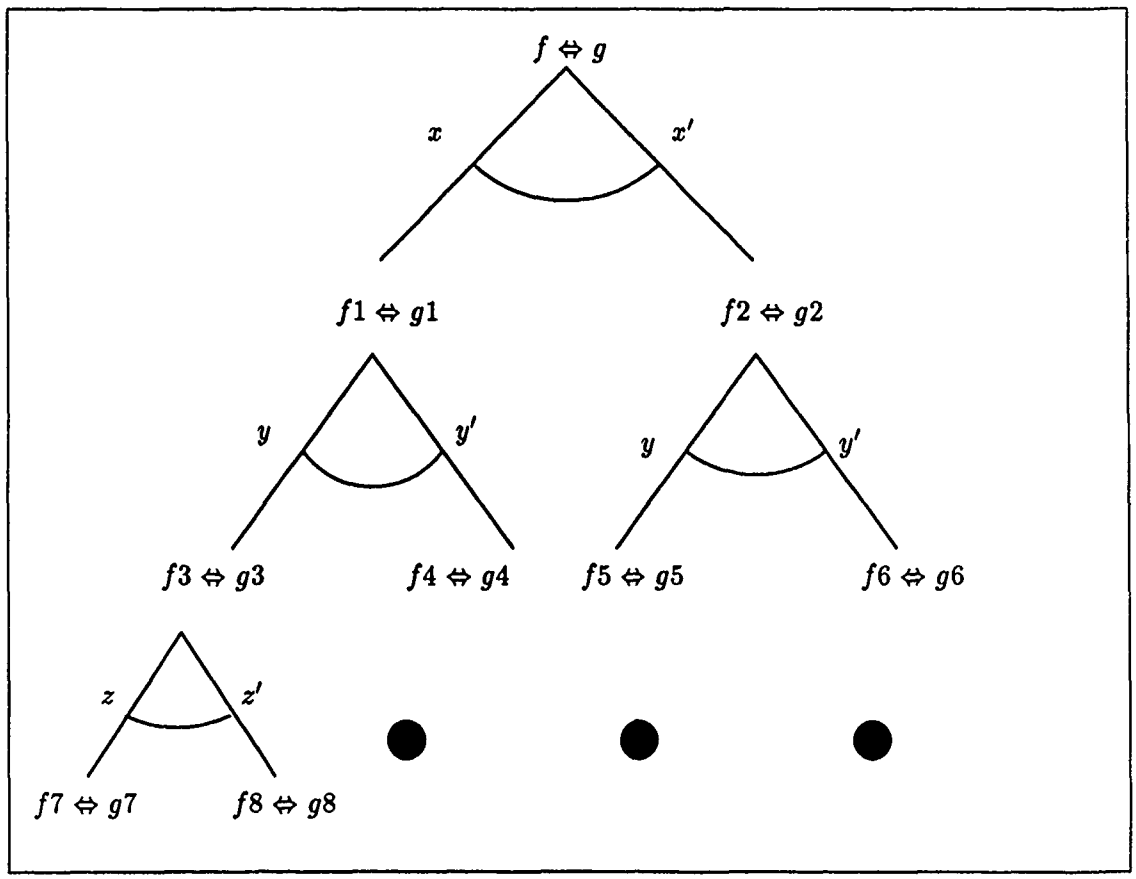

Figure 4.1. Boole's Expansion AND-tree.

Since the search space is an AND-tree, once a branch is found to be false, the search can terminate with failure. A blind, depth-first search can be performed by simply selecting the first variable and substituting logical 0s and 1s for it. A general best-first search technique can be used by creating some criterion for the best variable to choose, such as most occurrences in function f or g or both. Still, in the worst case, the tree will have to expand 2 AND-nodes at each level to a depth of n variables. This results in $2^n - 1$ expansions, but in most cases, due to pattern matching and equivalence checking at each expansion, fails or succeeds much faster.

CPT Dukes's code has been modified to work for the type of structures used in AFIT_VERIFY. The code used by AFIT_VERIFY will now be summarized. A more detailed discussion is provided in the reference (6:39–43).

CPT Dukes defined **xor**, **or**, **and**, and **not** as operators represented by the respective one-character symbols \$,@,^, and ~. In AFIT_VERIFY, **xor**, **or**, **and**, and **neg** are used as principal functors. The **eval** clauses are Boolean simplifications of the functors xor, or, and, and **neg**.

```
eval(or(1,_),1):-!.
eval(and(1,X),X):-!.
eval(xor(1,X),neg(X)):-!.
eval(or(0,X),X):-!.
eval(and(0,_),0):-!.
eval(xor(0,X),X):-!.
eval(or(_,1),1):-!.
eval(and(X,1),X):-!.
eval(xor(X,1),neg(X)):-!.
eval(or(X,0),X):-!.
eval(and(_,0),0):-!.
eval(xor(X,0),X):-!.
eval(neg(0),1):-!.
eval(neg(1),0):-!.
eval(or(neg(X),X),1):-!.
eval(or(X,neg(X)),1):-!.
eval(and(neg(X),X),0):-!.
eval(and(X,neg(X)),0):-!.
eval(xor(X,X),0):-!.
eval(xor(neg(X),X),1):-!.
```

```
eval(xor(X,neg(X)),1):-!.
eval(or(X,X),X):-!.
eval(and(X,X),X):-!.
eval(X,X):-!.
```

The **extract** clauses are used to extract a variable from a structure to use in the Boolean expansion. (These currently perform a blind, depth-first search.)

```
extract(X,X) :-                    /* X is of the form inX(Avariable) */
  X =.. [_,Arg],
  var(Arg),!.
extract(X,neg(Y)) :-
  extract(X,Y).
extract(X,or(L,_)) :-
  extract(X,L).
extract(X,or(_,R)) :-
  extract(X,R).
extract(X,and(L,_)) :-
  extract(X,L).
extract(X,and(_,R)) :-
  extract(X,R).
extract(X,xor(L,_)) :-
  extract(X,L).
extract(X,xor(_,R)) :-
  extract(X,R).
```

The clauses **remove_x_1** and **remove_x_0** replace every occurrence of the variable x with the Boolean value 1 and 0, respectively. The two clauses are identical, except for the Boolean value replaced; therefore, it is only necessary to include one of the clauses for explanatory purposes.

```
remove_x_1(Y,X,Y) :-
  atomic(Y),!.
remove_x_1(neg(Y),X,neg(Y)) :-
  atomic(Y),!.
```

```
remove_x_1(Y,X,Y) :-
  Y =.. [I1,Arg1],
  var(Arg1),
  X =.. [I2,Arg2],                /* we already know X is a Variable */
  I1 \== I2,!.
remove_x_1(neg(Y),X,neg(Y)) :-
  Y =.. [I1,Arg],
  var(Arg1),
  X =.. [I2,Arg2],                /* we already know X is a Variable */
  I1 \== I2,!.
remove_x_1(Y,X,1):-
  Y =.. [I1,Arg1],                 /*  in0(_1) \== in0(_2) in Prolog */
  var(Arg1),                      /* but we know it is the same input */
  X =.. [I1,Arg2],!.               /* we already know X is a Variable */
remove_x_1(neg(Y),X,0):-
  Y =.. [I1,Arg1],
  var(Arg1),
  X =.. [I1,Arg2],!.              /* we already know X is a Variable */
remove_x_1(neg(Y),X,neg(NewY)) :-
  !,remove_x_1(Y,X,NewY).
remove_x_1(or(L,R),X,or(LNew,RNew)) :-
  !,remove_x_1(L,X,LNew),
  remove_x_1(R,X,RNew).
remove_x_1(and(L,R),X,and(LNew,RNew)) :-
  !,remove_x_1(L,X,LNew),
  remove_x_1(R,X,RNew).
remove_x_1(xor(L,R),X,xor(LNew,RNew)) :-
  !,remove_x_1(L,X,LNew),
  remove_x_1(R,X,RNew).
```

The **divide** clause is used to expand the clause, F, into two clauses, one where every X is replaced by logical 0 and the other where every X is replaced by logical 1. The resulting functions are then simplified by the **evaluate_dukes** clauses.

```
divide(F,X,F0,F1) :-
  remove_x_0(F,X,F0Temp),
  remove_x_1(F,X,F1Temp),
  evaluate_dukes(F0Temp,F0),
  evaluate_dukes(F1Temp,F1).
```

```
evaluate_dukes(X,X) :-
  atomic(X),!.
evaluate_dukes(X,X):-
  X =.. [I,Arg],
  var(Arg),
  I \== neg,!.
evaluate_dukes(neg(F),FReduced) :-
  evaluate_dukes(F,FTemp),
  eval(neg(FTemp),FReduced),!.
evaluate_dukes(or(L,R),Resolved) :-
  evaluate_dukes(L,LNew),
  evaluate_dukes(R,RNew),
  eval(or(LNew,RNew),Resolved).
evaluate_dukes(and(L,R),Resolved) :-
  evaluate_dukes(L,LNew),
  evaluate_dukes(R,RNew),
  eval(and(LNew,RNew),Resolved).
evaluate_dukes(xor(L,R),Resolved) :-
  evaluate_dukes(L,LNew),
  evaluate_dukes(R,RNew),
  eval(xor(LNew,RNew),Resolved).
```

The eq clauses are the drivers for the Boolean expansion. First, the eq clauses check for trivial equivalence of functions F and G. If functions F and G are not equivalent, eq extracts a variable X, replaces it with 1s and 0s in both functions F and G to produce the new functions, F0, F1, G0, and G1. The process is then repeated on the new functions. Checking for equivalence at each stage greatly enhances the chance of producing trivial equivalence or failure, and results in a faster search than performing a complete Boolean expansion and then checking for equivalence only once.

```
eq(X,X):-!.
eq(F,G) :-
  extract(X,F),
  divide(F,X,F0,F1),
  divide(G,X,G0,G1),!,
  eq(F0,G0),!,
  eq(F1,G1),!.
```

4.6 Summary

AFIT_VERIFY provides a framework to verify any type of hardware module (i.e., modules with multiple outputs and/or multiple states) where feedback is broken by a register. AFIT_VERIFY was developed in a modular fashion to allow the addition of new behaviors. When a new behavior is required, only `derive_behavior`, `evaluate_brown`, and `replace_all` clauses need to be added. No modification of existing code should be required. The portion of code which determines the equivalence of the derived outputs/states and specified outputs/states will need to be expanded. AFIT_VERIFY currently only performs simple canonicalizations, trivial equivalence, and Boolean expansion; therefore, only Boolean or simply canonicalized modules can be verified. All code was developed on a PC using PROLOG-1, since this provided the best prototype development available at the time. PROLOG-1 was not powerful enough to verify a full-adder module, due to stack overflow. Work continued on mainframe Quintus Prolog, which had no problem with the full-adder verification. Future work at AFIT is expected to be performed on a PC version of Quintus Prolog.

V. Results and Recommendations

5.1 Results

The goal of this thesis was to provide a Prolog implementation of the concepts discussed by Barrow (1). AFIT_VERIFY would enable an AFIT engineering student to describe the behavior and structure of a logic circuit. AFIT_VERIFY would then derive a behavioral description from the structural description and determine if the two behavioral descriptions are equivalent. This goal has been met. An AFIT engineering student can describe, in Prolog, combinational and sequential logic circuits composed of nand2, xor, **full-adder**, **mux**, **inc**, **reg**, and **counter** modules. Behavior must be specified by using NAND, XOR, full-adder, AND, OR, neg, inc, MUX, reg, and counter structures in Prolog functor notation as shown in in the examples in Appendix A. The circuit descriptions currently created are the following;

1. **nand2** - A two input NAND gate (Primitive component).
2. **xor** - A two input XOR gate composed of nand2 submodules.
3. **faddxor** - A full-adder composed of xor and nand2 submodules.
4. **mux** - A 2X1 multiplexer (Primitive Component).
5. **reg** - A register which stores an integer (Primitive Component).
6. **inc** - An integer incrementer (Primitive Component).
7. **counter** - An integer counter composed of mux, reg, and inc submodules.

Two non-primitive modules, the **counter** and **faddxor** modules have been verified using AFIT_VERIFY. Sample runs are included as Appendix B. The **counter** module provides an example of a simple sequential circuit. The **counter** module is composed entirely of primitive components, but demonstrates AFIT_VERIFY's ability to treat a circuit as a black-box. The behavior of the external state **count** is derived by deriving the state of its internal representation, the **contents** of the register. Then, **count** is substituted for **contents** wherever it may appear in the derived behavior. The **counter** module was

verified on PROLOG-1 and Quintus Prolog. Two versions of AFIT_VERIFY were used. One version contains outputs which helped in debugging the development code. The second version removed all ouputs to show how much time is devoted to I/O. The results for the `counter` and `faddxor` modules are summarized in Table 5.1.

Table 5.1. PROLOG-1 vs. Quintus Prolog Run Times

Module	PROLOG-1	Quintus Prolog
`counter(outputs)`	11.0*sec*	2.0*sec*
`counter(no outputs)`	5.0*sec*	1.0*sec*
`faddxor(outputs)`	175.0*sec*	13.0*sec*
`faddxor(no outputs)`	83.0*sec*	7.0*sec*

Quintus Prolog was much faster, as expected. The times are encouraging for further work. The `faddxor` module, as specified, contains 11 primitive gates (nand2). This is not a significant number of gates, but it requires the hierarchical verification of the `xor` module in the process of verifying the `faddxor` module. The derivation ultimately contains 30 connections; 12 from the `faddxor` specification and 9 each from the 2 `xor` specifications. Though not overwhelming evidence, it suggests that circuits with 1000 structural connections could be verified in a few minutes. The AFIT_VERIFY methodology is definitely deserving of more research. It should be noted that the `faddxor` module could not be verified by PROLOG-1 with the "output" version of AFIT_VERIFY. This was due to insufficient stack space.

Currently, all modules, submodules, and the verification code need to be loaded in the order specified in Appendix A prior to verification of a module. The types of modules which should exist as primitives and be loaded with the verification code will need to be determined in follow-on work. Since the examples were sufficiently small for mainframe Quintus Prolog, no system state (`verified, derived_behavior, next_state` clauses) had to be written to disk for possible system restart (1:487). When larger circuits (1000 internal connections) are verified, a method for saving system state and restarting verification from that point will be required. Prolog does provide built-in mechanisms to perform state savings during the course of a program.

Each Prolog clause was developed and tested modularly by first ensuring the proper operation of each subclause and then testing each clause. The majority of the code was developed in a three month period. The code is not verified, but each clause does provide the expected behavior. When AFIT_VERIFY claims that a module is verified, it can be trusted. AFIT_VERIFY provides AFIT with a method to provide true module verification. AFIT_VERIFY is only a start towards a complete design/verification system, but work towards such a system will continue at AFIT. This research should result in great time and cost savings to AFIT and possibly to the rest of the engineering community.

5.2 Recommendations

Many different areas can be researched in follow-on efforts. The following is only a suggested list.

1. Verification of Gordon's D74 (1:456) or other larger circuit.
2. In conjunction with the AFIT VLSI group, integration of AFIT_VERIFY into a larger design system similar to Grabowiecki et al. (10) .
3. Integration into AFIT_VERIFY of a Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) to Prolog parser for the construction of Prolog behavioral and structural specifications . (A VHDL to Prolog parser exists on the AFIT Microvax Cub under the directory, ~ges/MCNC/vhdl2.)
4. Analysis of Prolog vs HOL or Prolog/HOL hybrid approach.

The first thing to consider before any of these areas can be researched is the viability of a Prolog approach. The specifications, structural and behavioral, required for small circuits are small, but the size of a structural specification is a function of the number of connections, or connected facts. If behavior and structural equivalence (i.e. low-level behavioral specification) can be modelled at the gate level, and it should be, a Prolog approach appears feasible. This feasibility is based on the Quintus Prolog run times discussed above. If all connections are required at the transistor level, then it could take years to verify a large circuit using Prolog.

Another problem is, " What is behavior, and how can it be specified?" It is straightforward for a combinational logic circuit composed of standard gates. But how does one represent a sequential circuit with feedback not broken by a state variable. This problem is expected to be studied in the next thesis cycle.

After consultation with the AFIT VLSI group, it appears that future research should proceed as follows: Obtain a circuit from the AFIT VLSI group containing 100 to 200 connections. Verify this circuit using AFIT_VERIFY, adding the appropriate behavioral constructs common to the AFIT VLSI group. (This will be performed in the next thesis cycle.) If this proves successful (i.e., verification requires minutes as opposed to hours), a larger (more than 1000 connections) circuit should be identified for verification. Integral to this phase should be the incorporation of the previously mentioned VHDL to Prolog parser. This would provide the AFIT VLSI group with the capability to use the DoD-mandated HDL. Integration of the Prolog verifier into a larger VLSI design system should be the ultimate goal, but some verification capability is preferred to no verification capability. The ability to verify portions of a large circuit can provide an increased capability to the VLSI group. One must also remember that AFIT_VERIFY is a hierarchichal verification system. Verification of one module of the circuit at a time is the method used to speed up the verification process. It is also the method which should be used in further developing the system.

Appendix A. AFIT_VERIFY Code

The following files constitute the development code for AFIT_VERIFY. The majority of code was developed using PROLOG-1. The only differences in PROLOG-1 and Quintus Prolog code are handled in the two separate files qops.pro (Quintus) and ops.pro (PROLOG-1). All other files remain the same on both systems. The files developed for AFIT_VERIFY are the following:

1. qops.pro - Quintus Prolog operator definitions and system-dependent procedures.
2. ops.pro - PROLOG-1 operator definitions and system-dependent procedures.
3. boole2.pro - CPT Duke's Modified Boolean Expansion code.
4. eval.pro - Rudimentary canonicalization clauses.
5. derbeh.pro - Derive Module behavior.
6. derstate.pro - Derive Module next state.
7. eqbeh.pro - Determine Behavior Equivalence.
8. verify.pro - Driver clause.
9. counter.pro - Counter, Incrementer, Register, and Multiplexer Module specifications.
10. xor.pro - Exclusive-OR and NAND Module specifications.
11. faddxor.pro - Full Adder Module specification.

For use in LaTeX, the .pro files were slightly modified and changed to .tex files. The .pro files were given to Dr. Brown on a $5\frac{1}{4}'', 360K$ floppy disk and also appear on the AFIT Microvax System Cub under `~ksparks\thesis`.

```
/************************************************************ */
/*                        QOPS.PRO                            */
/*  This file provides the utility and operator definitions */
/*  to run verify.pro on Quintus Prolog.  The :- multifile  */
/*  definition was required to allow Modules to be declared */
/*  in separate .pro files.  Without this definition, any   */
/*  new module loaded would wipe out the previously loaded  */
/*  modules, and in the case of a fulladder (multiple file) */
/*  this was a problem (the defn of nand2 and xor was gone) */
/*  Operator precedence in Quintus goes from 0-1200, but    */
/*  PROLOG-1 runs 0-255 , PROLOG-1 refman 5.5, Bratko p.182 */
/*                                                          */
/************************************************************/

:- multifile module_name/1, port/4, part/3, output_eqn/2, state_eqn/2,
   state_map/3, state_of/3, connected/3.

/*--------------------- UTILITIES ------------------------- */

writeln([]) :-
   nl.
writeln([X|Rest]) :-
   write(X),
   writeln(Rest).
/*****************************************************************/
/*                                                               */
/*  The goal 'find_all(T,G,L)' constructs a list L consisting    */
/*  of all instantiations of the term T for which the goal G     */
/*  is satisfied. The goal                                       */
/*       find_all([X,wife_of,Y], husband(Y,X), Couples) ,        */
/*  for example, might instantiate Couples to                    */
/*   [[amy,wife_of,tony],[sue,wife_of,john],[dot,wife_of,tom]]   */
/*  for a given family-database. See the Bratko text, p. 177,    */
/*  for another implementation of 'find_all'.                    */
/*                                                               */
/*  FIND_ALL is needed to check for proper operation of the      */
/*  clauses verify_components, derive_and_equate_behaviors, and */
/*  derive_and_equate_states.                                    */
/*---------------------------------------------------------------*/

find_all(T,G,L) :-              /* To make a list L of terms T  */
  store_elts(T,G),              /* satisfying the goal G, store */
  build_list([],L).             /* the terms as facts and then  */
                                /* collect the facts in an      */
                                /* initially-empty list.        */
```

```
store_elts(T,G) :-              /* 'store(T,G)' has the side-   */
  asserta(element(end)),        /* effect of forming a sequence */
  call(G),                      /* of facts having the form     */
  asserta(element(T)),          /* 'element(T)', where T is a   */
  fail.                         /* term satisfying the goal G.  */
store_elts(_,_).                /* The second clause enables    */
                                /* the procedure to succeed.    */

build_list(Part,Total) :-       /* 'build_list(P,Q)' succeeds   */
  get_next(T),                  /* in case Q is the list re-    */
  !,                            /* resulting from attaching the */
  build_list([T|Part],Total).   /* 'element' facts in the cur-  */
build_list(L,L).                /* rent database to the list P. */

get_next(T) :-
  retract(element(T)),
  !,
  T \== end.

length_of([],0).                /* Bratko, p. 88                */
length_of([_|L],N) :-
   length_of(L,N1),
   N is N1 + 1.

member(X,[X|_]).                /* Bratko, p. 68                */
member(X,[_|L]) :-
   member(X,L).

undup([],[]).                   /* The null list contains no    */
undup([X|T],M) :-               /* duplication.  If X is in the */
   member(X,T),                 /* tail, then remove all dups from */
   !,                           /* the tail.  If X is not in the */
   undup(T,M).                  /* tail, return X with other dups */
undup([X|T],[X|M]) :-           /* removed from the tail.       */
   undup(T,M).
```

```
/*************************************************************/
/*                                                           */
/*  SET_OF performs the same function as FIND_ALL, except    */
/*   all duplicates are removed from the list L. SET_OF is   */
/*   not defined in PROLOG-1 but SETOF is defined in         */
/*   However; since SETOF didn't perform as expected, the    */
/*   clause SETOF is also used in QUINTUS PROLOG.            */
/*                                                           */
/*  The above utilities member(X,L) and undup(L,UL) are      */
/*   required to produce set_of(T,G,UL). The member(X,L)      */
/*   clause determines if X is a member of list L. The       */
/*   undup(L,UL) clause removes duplicates from list L to    */
/*   produce list UL.                                        */
/*                                                           */
/*************************************************************/
set_of(T,G,UL) :-
 find_all(T,G,L),
 undup(L,UL).


/*--------------------- OPERATORS ------------------------- */

?- unknown(trace,fail).
?- op(100, fy, not).

not X :-
  call(X),!,fail;
  true.

?- op(900, xfx, :=).
?- op(800, fx, if).
```

```
/*************************************************************/
/*                        OPS.PRO                            */
/*  This file provides the utility and operator definitions*/
/*  to use verify.pro in Prolog-1.  Remember, in Prolog-1  */
/*  operator precedence runs from 0 to 255.                */
/*************************************************************/

/*--------------------- UTILITIES -------------------------*/

writeln([]) :-
   nl.
writeln([X|Rest]) :-
   write(X),
   writeln(Rest).

/*********************************************************************/
/*                                                                   */
/*  The goal 'find_all(T,G,L)' constructs a list L consisting        */
/*  of all instantiations of the term T for which the goal G         */
/*  is satisfied. The goal                                           */
/*       find_all([X,wife_of,Y], husband(Y,X), Couples) ,            */
/*  for example, might instantiate Couples to                        */
/*   [[amy,wife_of,tony],[sue,wife_of,john],[dot,wife_of,tom]]       */
/*  for a given family-database. See the Bratko text, p. 177,        */
/*  for another implementation of 'find_all'.                        */
/*                                                                   */
/*  FIND_ALL is needed to check for proper operation of the          */
/*  clauses verify_components, derive_and_equate_behaviors, and      */
/*  derive_and_equate_states.                                        */
/*-------------------------------------------------------------------*/

find_all(T,G,L) :-              /* To make a list L of terms T  */
  store_elts(T,G),              /* satisfying the goal G, store */
  build_list([],L).             /* the terms as facts and then  */
                                /* collect the facts in an      */
                                /* initially-empty list.        */

store_elts(T,G) :-              /* 'store(T,G)' has the side-   */
  asserta(element(end)),        /* effect of forming a sequence */
  call(G),                      /* of facts having the form     */
  asserta(element(T)),          /* 'element(T)', where T is a   */
  fail.                         /* term satisfying the goal G.  */
store_elts(_,_).                /* The second clause enables    */
                                /* the procedure to succeed.    */
```

```
build_list(Part,Total) :-       /* 'build_list(P,Q)' succeeds   */
  get_next(T),                  /* in case Q is the list re-    */
  !,                            /* resulting from attaching the */
  build_list([T|Part],Total).   /* 'element' facts in the cur-  */
build_list(L,L).                /* rent database to the list P. */

get_next(T) :-
  retract(element(T)),
  !,
  T \== end.

length_of([],0).                /* Bratko, p. 88                */
length_of([_|L],N) :-
   length_of(L,N1),
   N is N1 + 1.

member(X,[X|_]).                /* Bratko, p. 68                */
member(X,[_|L]) :-
   member(X,L).

undup([],[]).                   /* The null list contains no    */
undup([X|T],M) :-               /* duplication.  If X is in the */
   member(X,T),                 /* tail, then remove all dups from */
   !,                           /* the tail.  If X is not in the */
   undup(T,M).                  /* tail, return X with other dups */
undup([X|T],[X|M]) :-           /* removed from the tail.       */
   undup(T,M).
```

```
/*************************************************************/
/*                                                           */
/*  SET_OF performs the same function as FIND_ALL, except    */
/*   all duplicates are  removed from the list L. SET_OF     */
/*   is not defined in PROLOG-1 but is defined in QUINTUS    */
/*   PROLOG.                                                 */
/*                                                           */
/*  The above utilities member(X,L) and undup(L,UL) are      */
/*   required to produce set_of(T,G,UL).  The member(X,L)    */
/*   clause determines if X is a member of list L.  The      */
/*   undup(L,UL) clause removes duplicates from list L to    */
/*   produce list UL.                                        */
/*                                                           */
/*************************************************************/
set_of(T,G,UL) :-
 find_all(T,G,L),
 undup(L,UL).

/*--------------------- OPERATORS -------------------------*/

?- op(200, xfx, :=).
?- op(190, fx, if).
```

```
/*************************************************************/
/*                                                           */
/*                    BOOLE'S EXPANSION                      */
/*                                                           */
/*  The following code is a modified version of code         */
/*  developed by CPT Mike Dukes.                             */
/*                                                           */
/*  He defined xor, or, and, and not as the respective       */
/*  operators, $, @, ^, and ~.  I use xor, or, and, and      */
/*  neg as principal functors.                               */
/*  A detailed explanation of this code can be found in      */
/*  CPT Dukes's PhD Prospectus, ''Formal Verification        */
/*  Using VHDL''.  Additional in-line comments are           */
/*  provided where modification for this approach is         */
/*  required.                                                */
/*                                                           */
/*************************************************************/

eval(or(1,_),1):-!.
eval(and(1,X),X):-!.
eval(xor(1,X),neg(X)):-!.
eval(or(0,X),X):-!.
eval(and(0,_),0):-!.
eval(xor(0,X),X):-!.
eval(or(_,1),1):-!.
eval(and(X,1),X):-!.
eval(xor(X,1),neg(X)):-!.
eval(or(X,0),X):-!.
eval(and(_,0),0):-!.
eval(xor(X,0),X):-!.
eval(neg(0),1):-!.
eval(neg(1),0):-!.
eval(or(neg(X),X),1):-!.
eval(or(X,neg(X)),1):-!.
eval(and(neg(X),X),0):-!.
eval(and(X,neg(X)),0):-!.
eval(xor(X,X),0):-!.
eval(xor(neg(X),X),1):-!.
eval(xor(X,neg(X)),1):-!.
eval(or(X,X),X):-!.
eval(and(X,X),X):-!.
eval(X,X):-!.
```

```
extract(X,X) :-                          /* X is of the form inX(Avariable) */
  X =.. [_,Arg],
  var(Arg),!.
extract(X,neg(Y)) :-
  extract(X,Y).
extract(X,or(L,_)) :-
  extract(X,L).
extract(X,or(_,R)) :-
  extract(X,R).
extract(X,and(L,_)) :-
  extract(X,L).
extract(X,and(_,R)) :-
  extract(X,R).
extract(X,xor(L,_)) :-
  extract(X,L).
extract(X,xor(_,R)) :-
  extract(X,R).

remove_x_1(Y,X,Y) :-
  atomic(Y),!.
remove_x_1(neg(Y),X,neg(Y)) :-
  atomic(Y),!.
remove_x_1(Y,X,Y) :-
  Y =.. [I1,Arg1],
  var(Arg1),
  X =.. [I2,Arg2],                       /* we already know X is a Variable */
  I1 \== I2,!.
remove_x_1(neg(Y),X,neg(Y)) :-
  Y =.. [I1,Arg],
  var(Arg1),
  X =.. [I2,Arg2],                       /* we already know X is a Variable */
  I1 \== I2,!.
remove_x_1(Y,X,1):-
  Y =.. [I1,Arg1],                        /*  in0(_1) \== in0(_2) in Prolog */
  var(Arg1),                             /* but we know it is the same input */
  X =.. [I1,Arg2],!.                      /* we already know X is a Variable */
remove_x_1(neg(Y),X,0):-
  Y =.. [I1,Arg1],
  var(Arg1),
  X =.. [I1,Arg2],!.                      /* we already know X is a Variable */
remove_x_1(neg(Y),X,neg(NewY)) :-
  ! remove_x_1(Y,X,NewY).
remove_x_1(or(L,R),X,or(LNew,RNew)) :-
  !,remove_x_1(L,X,LNew),
  remove_x_1(R,X,RNew).
```

```
remove_x_1(and(L,R),X,and(LNew,RNew)) :-
  !,remove_x_1(L,X,LNew),
  remove_x_1(R,X,RNew).
remove_x_1(xor(L,R),X,xor(LNew,RNew)) :-
  !,remove_x_1(L,X,LNew),
  remove_x_1(R,X,RNew).

remove_x_0(Y,X,Y) :-
  atomic(Y),!.
remove_x_0(neg(Y),X,neg(Y)) :-
  atomic(Y),!.
remove_x_0(Y,X,Y) :-
  Y =.. [I1,Arg1],
  var(Arg1),
  X =.. [I2,Arg2],                    /* we already know X is a Variable */
  I1 \== I2,!.
remove_x_0(neg(Y),X,neg(Y)) :-
  Y =.. [I1,Arg1],
  var(Arg1),
  X =.. [I2,Arg2],                    /* we already know X is a Variable */
  I1 \== I2,!.
remove_x_0(Y,X,0):-
  Y =.. [I1,Arg1],
  var(Arg1),
  X =.. [I1,Arg2],!.                  /* we already know X is a Variable */
remove_x_0(neg(Y),X,1):-
  Y =.. [I1,Arg1],
  var(Arg1),
  X =.. [I1,Arg2],!.                  /* we already know X is a Variable */
remove_x_0(neg(Y),X,neg(NewY)) :-
  !,remove_x_0(Y,X,NewY).
remove_x_0(or(L,R),X,or(LNew,RNew)) :-
  !,remove_x_0(L,X,LNew),
  remove_x_0(R,X,RNew).
remove_x_0(and(L,R),X,and(LNew,RNew)) :-
  !,remove_x_0(L,X,LNew),
  remove_x_0(R,X,RNew).
remove_x_0(xor(L,R),X,xor(LNew,RNew)) :-
  !,remove_x_0(L,X,LNew),
  remove_x_0(R,X,RNew).
```

```
divide(F,X,F0,F1) :-
  remove_x_0(F,X,F0Temp),
  remove_x_1(F,X,F1Temp),
  evaluate_dukes(F0Temp,F0),
  evaluate_dukes(F1Temp,F1).


evaluate_dukes(X,X) :-
  atomic(X),!.
evaluate_dukes(X,X):-
  X =.. [I,Arg],
  var(Arg),
  I \== neg,!.
evaluate_dukes(neg(F),FReduced) :-
  evaluate_dukes(F,FTemp),
  eval(neg(FTemp),FReduced),!.
evaluate_dukes(or(L,R),Resolved) :-
  evaluate_dukes(L,LNew),
  evaluate_dukes(R,RNew),
  eval(or(LNew,RNew),Resolved).
evaluate_dukes(and(L,R),Resolved) :-
  evaluate_dukes(L,LNew),
  evaluate_dukes(R,RNew),
  eval(and(LNew,RNew),Resolved).
evaluate_dukes(xor(L,R),Resolved) :-
  evaluate_dukes(L,LNew),
  evaluate_dukes(R,RNew),
  eval(xor(LNew,RNew),Resolved).


eq(X,X):-!.
eq(F,G) :-
  extract(X,F),
  divide(F,X,F0,F1),
  divide(G,X,G0,G1),!,
  eq(F0,G0),!,
  eq(F1,G1),!.
```

```
/*************************************************************/
/*                        EVAL.PRO                           */
/*   This file performs a rudimentary simplification and     */
/*   and canonicalization on behavioral structures.          */
/*   Any new behavioral structures added will require        */
/*   additional evaluate_brown clauses.                      */
/*                                                           */
/*************************************************************/

evaluate1(X,EX) :-
   writeln(['Value of ',X,':']),
   evaluate_brown(X,EX),
   writeln(['          ',EX]).

/*************************************************************/
/*                                                           */
/*  The first three clauses provide basis cases for a        */
/*  behavioral structure, namely a Variable, Atom, or an     */
/*  elementary structure.                                    */
/*                                                           */
/*************************************************************/

evaluate_brown(X,X) :-
   var(X),!.
evaluate_brown(X,X) :-
   atomic(X),!.
evaluate_brown(Struct,Struct) :-
   Struct =.. [F,Arg],
   ( var(Arg) ; atom(Arg) ),!.
```

```
/*********************************  **********************/
/*                                                       */
/*  The next three clauses provide a method of simplifying  */
/*  and canonicalizing the boolean functions and, or,        */
/*  and negation. Another canonical form may prove quicker  */
/*  and these clauses would need to be modified accordingly.*/
/*                                                       */
/*  NOTE:  neg has been chosen as the negation functor, as  */
/*  opposed to the more widely used not, since Prolog-1      */
/*  defines not as the absence of a fact.  The functor not  */
/*  is not provided in Pure Prolog and therefore, does not  */
/*  exist in Quintus Prolog.  This is why I have defined not*/
/*  in the qops.pro file.                                    */
/*                                                       */
/*********************************************************/

evaluate_brown(and(X,Y),Value) :-
  evaluate_brown(X,EX),
  evaluate_brown(Y,EY),
  (  (EX  = 0 ;  EY = 0),  !,
     Value = 0
  ;
     EX = 1, !,
     Value = EY
  ;
     EY = 1, !,
     Value = EX
  ;
     !,Value = and(EX,EY)
  ).

evaluate_brown(or(X,Y),Value) :-
   evaluate_brown(X,EX),
   evaluate_brown(Y,EY),
   (  (EX = 1 ;  EY = 1), !,
      Value = 1
   ;
       EX = 0, !,
       Value = EY
   ;
       EY = 0,  !,
       Value = EX
   ;
       !,Value = or(EX,EY)
   ).
```

```
evaluate_brown(neg(X),Value)  :-
   evaluate_brown(X,EX),
   (  var(EX), !,
      Value = neg(X)
   ;
      EX = 0,  !,
      Value = 1
   ;
      EX = 1, !,
      Value = 0
   ;
      atom(EX), !,
      Value = neg(EX)
   ;
      EX = neg(N), !,
      Value = N
   ;
      EX = and(A1,A2),  !,
      evaluate_brown(neg(A1),NA1),
      evaluate_brown(neg(A2),NA2),
      Value = or(NA1,NA2)
   ;
      EX = or(O1,O2), !,
      evaluate_brown(neg(O1),NO1),
      evaluate_brown(neg(O2),NO2),
      Value = and(NO1,NO2)
   ;
      !,Value = neg(EX)
   ).
```

```
/*************************************************************/
/*                                                           */
/*  The remaining clauses provide a method of simplification*/
/*  and canonicalization of the functions required to verify*/
/*  the fulladder and counter, namely, if and +.             */
/*                                                           */
/*************************************************************/

evaluate_brown(if(Cond,Texp,Fexp),Value) :-
   evaluate_brown(Cond,NCond),
   evaluate_brown(Texp,NTexp),
   evaluate_brown(Fexp,NFexp),
   (  ( NCond = 1, !,                      /* Condition is true */
        Value = NTexp ) ;                  /*  return True exp  */
      ( NCond = 0, !,                      /* If False then     */
        Value = NFexp ) ;                  /*  return False exp */
      ( NTexp = NFexp, !,                /* Condition irrelevant */
        Value = NTexp ) ;                  /*  if choices equal */
      Value = if(NCond,NTexp,NFexp),!).    /* otherwise return */
                                        /* simplified expression. */
evaluate_brown(X+Y,Z) :-
   integer(X),
   integer(Y), !,
   Z is X + Y.            /* force simplification of 1 + 2 = 3 */
evaluate_brown(X+Y,Z) :-
   integer(Y),!,                    /* X not integer due to cut */
   evaluate_brown(X,NewX),
    Z = Y + NewX.           /* canonicalize with integer first */
evaluate_brown(X+Y,Z) :-
   !,evaluate_brown(X,NewX),
   evaluate_brown(Y,NewY),
   Z = NewX + NewY.
evaluate_brown(X,X).    /*  default simplification for complex */
                        /*  structures like in(incA(X)).       */
```

```
/*************************************************************/
/*                    Derive_Behaviors                       */
/*                                                           */
/*    The two derive_behaviors clauses identify a specific   */
/*  output for the Module.  The derive_behavior clauses      */
/*  are then invoked to derive the behavior of that output   */
/*  for this particular Module.                              */
/*    The derive_behavior clause uses the part, connected,   */
/*  and output_eqn clauses to derive an output's behavior    */
/*  and tie it to this instantiation of the module as        */
/*  described in the in-line comments to follow.             */
/*    The derive_and_equate_behaviors clause in verify.pro   */
/*  uses derive_behaviors to derive all Module outputs and   */
/*  determine their equivalence to the specified output.     */
/*    The arguments for derive_behaviors and derive_behavior*/
/*  have the following meanings:                             */
/*                                                           */
/*  Args:  Module:  e.g., 'xor','nand2', ...                 */
/*         Form:  A formula involving terminal-behavior.     */
/*                  In the initial query, this may be        */
/*                  something like 'out(X)'.                 */
/*         Behavior: The resulting derived behavior.         */
/*  In the present version of this procedure, it is assumed  */
/*  that all of the component-parts of Module have been      */
/*  previously verified by verify_components.  The verified  */
/*  components derived behavior is either asserted in a      */
/*  derived_behavior clause or specified in an output_eqn    */
/*  if the component-part is a primitive.                    */
/*                                                           */
/*************************************************************/


derive_behaviors(Module,Form,Behavior) :-                /*  no state */
   not state_eqn(Module,_),!,
   output_eqn(Module,Form := Spec_Behavior),
   derive_behavior(Module,Form,Behavior).
derive_behaviors(Module,Form,Behavior) :-             /* has state (!), */
   output_eqn(Module,Form := Spec_Behavior),
   derive_behavior(Module,Form,TBehavior),
   substitute_state(Module,TBehavior,Behavior). /* might  require the */
                              /* removal of some internal variables. */
```

```
/*************************************************************/
/* Rules 1A and 1B derive behavior if Form is the name of a  */
/* terminal to which some other terminal, in Module, is      */
/* connected.  Rule 1A is invoked if the other terminal is a */
/* primary terminal in Module, i.e., one of its inputs or    */
/* outputs. Rule 1B is invoked if the other terminal belongs */
/* to one of Module's component-parts.                       */
/*************************************************************/

derive_behavior(Module, Form, Source) :-
   connected(Module, Source, Form),
   primary_source(Source),!,
   writeln(['Applying Rule 1A to ',Form]).
derive_behavior(Module, Form, Behavior) :-
   connected(Module, Source, Form),
   derived_source(Source),!,
   writeln(['Applying Rule 1B to ', Form]),
   derive_behavior(Module, Source, Behavior).

/*************************************************************/
/*  primary_source and derived source distinguish between a  */
/*  Module input(primary_source) and a Component input       */
/*  (secondary_source).                                      */
/*************************************************************/
primary_source(Source) :-
   Source =.. [_,Arg],
   var(Arg).

derived_source(Source) :-
   Source =.. [_,Arg],
   Arg =.. [_,Arg2],
   var(Arg2).
```

```
/*****************************************************************/
/* Rule 2 is invoked if Form is the name of a terminal of one */
/* of Module's component-parts.   Rule 2A handles primitive   */
/* components where Rule 2B handles non-primitive components. */
/* The only real difference is where to locate the Components */
/* derived behavior  (output_eqn vs derived_behavior).  If it */
/* is later decided to assert a derived_behavior clause for   */
/*  primitives, then Rule 2A can be removed.                  */
/*****************************************************************/

derive_behavior(Module, Form, Behavior) :-
   Form \== 1,
   Form =.. [F,G],
   part(Module, G, Component),
   not part(Component,_,_),      /* Component is a primitive module */
   output_eqn(Component, Form := OutForm),!,
   writeln(['Applying Rule 2A to ', Form]),
   writeln([Component,'''s output equation:']),
   writeln(['   ', Form, ' := ', OutForm]),
   derive_behavior(Module, OutForm, Behavior).      /* replace gate */
                                  /* inputs with module variables  */
derive_behavior(Module, Form, Behavior) :-
   Form \== 1,
   Form =.. [F,G],
   part(Module, G, Component), /* from cut, not primitive component */
   /* previously verified due to verify_components in verify clause */
   derived_behavior(Component,Form,OutForm),!,
   writeln(['Applying Rule 2B to ', Form]),
   writeln([Component,'''s derived behavior:']),
   writeln(['   ', Form, ' := ', OutForm]),
   derive_behavior(Module,OutForm,Behavior).       /* replace gate */
                                  /* inputs with module variables  */
```

```
/*************************************************************/
/* The remaining rules cover cases in which FORM is not the  */
/* name of a terminal, but is a formula involving such name. */
/* This is where additional types of boolean or non-boolean  */
/* behavioral rules can be added in future work.  These rules*/
/* simplify internal components of a specified behavioral    */
/* structure.  The evaluate1 clause is a simple canonicalizer*/
/* which should also be modified if new behaviors are added. */
/*************************************************************/

derive_behavior(Module, neg(Form), Behavior) :-
   !,writeln(['Applying Rule 3 to ',neg(Form)]),
   derive_behavior(Module, Form, Beh1),
   evaluate1(neg(Beh1), Behavior).
derive_behavior(Module, and(Form1,Form2), Beh) :-
   !,writeln(['Applying Rule 4 to ',and(Form1,Form2)]),
   derive_behavior(Module, Form1, Beh1),
   derive_behavior(Module, Form2, Beh2),
   evaluate1(and(Beh1,Beh2), Beh).
derive_behavior(Module, or(Form1,Form2), Beh) :-
   !,writeln(['Applying Rule 5 to ',or(Form1,Form2)]),
   derive_behavior(Module, Form1, Beh1),
   derive_behavior(Module, Form2, Beh2),
   evaluate1(or(Beh1,Beh2), Beh).
derive_behavior(Module, if(Cond,Texp,Fexp), Beh) :-
   !,writeln(['Applying Rule 6 to ',if(Cond,Texp,Fexp)]),
   derive_behavior(Module, Cond, NCond),
   derive_behavior(Module, Texp, NTexp),
   derive_behavior(Module, Fexp, NFexp),
   evaluate1(if(NCond,NTexp,NFexp), Beh).
derive_behavior(Module, First + Second, Beh) :-
   !,writeln(['Applying Rule 7 to ',First + Second]),
   derive_behavior(Module, First, Beh1),
   derive_behavior(Module, Second, Beh2),
   evaluate1(Beh1 + Beh2, Beh).
/*************************************************************/
/* The default rule catches behavior which we haven't yet    */
/* described in a rule or which shouldn't be described.      */
/*************************************************************/
derive_behavior(Module, Form, Form) :-              /* default Rule */
   writeln(['Applying default Rule to ',Form]).
```

```
/*************************************************************/
/*                                                           */
/*                    DERSTATE.PRO                           */
/*                                                           */
/*  The clause derive_states finds a state variable          */
/*  for a Module, how this state variable fits into the      */
/*  internal structure of the Module, derives the behavior   */
/*  of the internal structure, and substitutes the           */
/*  state variable name for the internal name whenever it    */
/*  appears in the derived behavior.  The state_of,          */
/*  state_map, and state_eqn facts from the specified        */
/*  Module description are used to identify the              */
/*  appropriate variables. Then the two clauses              */
/*  derive_behavior and substitute_state are invoked to      */
/*  create the desired Next_State.                           */
/*                                                           */
/*************************************************************/

derive_states(Module,State,Next_State) :-
   state_of(Module,State,Type),                          /*  It has state  */
   state_map(Module,State,Internal), /* It's mapped to an internal part */
   state_eqn(Part,Internal := NextState/* internal state is function of */
   derive_behavior(Module,NextState,Beh),  /* inputs and previous state */
   substitute_state(Module,Beh,Next_State).

/*************************************************************/
/*                                                           */
/*                    substitute_state                       */
/*                                                           */
/*  This clause uses replace_all to replace occurrences      */
/*  of internal variables with the appropriate external      */
/*  black-box variable obtained by the state_map fact.       */
/*************************************************************/

substitute_state(Module,DerBeh,SubBeh) :-
   state_map(Module,External,Internal),
   !,writeln(['Derived Behavior:  ',DerBeh]),
   replace_all(Module,Internal,External,DerBeh,SubBeh),
   writeln(['Substituted Behavior:  ',SubBeh]).
```

```
/*************************************************************/
/*                                                           */
/*                  replace_all                              */
/*   Replaces each occurrence of in internal variable or     */
/*   other variables connected to this internal variable     */
/*   with the appropriate external balck-box variable.       */
/*   The replace clauses allow you to traverse any type      */
/*   of behavioral structure and replace the appropriate     */
/*   variable name.  New replace clauses will need to be     */
/*   added when new behavioral structures are created.       */
/*************************************************************/

replace_all(Module,Old,New,OldBeh,SubBeh) :-
   replace(Old,New,OldBeh,SB),
   (   connected(Module,Old,Other) ;
       output_eqn(Part,Other := Old) ),!,
   replace_all(Module,Other,New,SB,NewSB),
   evaluate1(NewSB,SubBeh).          /* Simplify further if possible */
replace_all(Module,Old,New,Beh,Beh) :-!.     /* no more connections */

replace(Old,New,Other,Other) :-
   atomic(Other),
   write('Rule2'),nl,!.
replace(Old,New,Other,Other) :-
   var(Other),
   write('Rule3'),nl,!.
replace(Old,New,Other,Other) :-
   Old =.. [F,Arg1],                  /* keeps in(X) = in(incA(X))  */
   Other =.. [G,Arg2],                /* from occuring, occurs test */
   F \== G,
   ( var(Arg2) ; atomic(Arg2) ),              /* already simplified */
   write('Rule4'),nl,!.
replace(Old,New,and(X,Y),and(NewB1,NewB2)) :-
   !,write('Rule and'),nl,
   replace(Old,New,X,NewB1),
   replace(Old,New,Y,NewB2).
replace(Old,New,or(X,Y),or(NewB1,NewB2)) :-
   !,write('Rule or'),nl,
   replace(Old,New,X,NewB1),
   replace(Old,New,Y,NewB2).
replace(Old,New,neg(X),neg(NewB)) :-
   !,write('Rule neg'),nl,
   replace(Old,New,X,NewB).
```

```
replace(Old,New,X + Y,NewB1 + NewB2) :-
   !,write('Rule + '),nl,
   replace(Old,New,X,NewB1),
   replace(Old,New,Y,NewB2).
replace(Old,New,if(Cond,Texp,Fexp),if(NewB1,NewB2,NewB3)) :-
   !,write('Rule if'),nl,
   replace(Old,New,Cond,NewB1),
   replace(Old,New,Texp,NewB2),
   replace(Old,New,Fexp,NewB3).
replace(Old,New,Other,NewB) :-              /* in(X) /== in(incA(X)) */
   Old =.. [F,Arg1],
   Other =.. [F,Arg2],
   ( ( var(Arg1), not var(Arg2) );
     ( not var(Arg1), var(Arg2) ) ),
   replace(Old,New,Arg2,NewArgs),
   NewB =.. [F,NewArgs],          /* Old behavior or Old Behavior is*/
   write('Rule struct'),nl,!.     /*  some other nested structure   */
replace(Old,New,Old,New) :-
   write('Rule 1'), nl,!.           /* If you find X replace with Y */
replace(Old,New,Other,Other) :-
    write('Default Rule'),nl.                    /* default rule.  */
```

```
/*************************************************************/
/*                                                           */
/*                        EQBEH.PRO                          */
/*  This file contains procedures necessary to determine     */
/*  the equivalence of a derived behavior(next state) and    */
/*  specified behavior(next state).                          */
/*  The equal_behaviors and equal_states clauses are used    */
/*  by derive_and_equate_behaviors and derive_and_equate_    */
/*  states to provide for every output/state equivalence.    */
/*  The primary methods of equivalence determination used    */
/*  are simiplification and boolean expansion.  The eqb      */
/*  clauses will require expansion in further work.          */
/*                                                           */
/*************************************************************/

equal_behaviors(Module,Output,Derived_Beh) :-
   output_eqn(Module,Output := Specified_Beh),/* get specified behavior */
   eqb(Module,Derived_Beh,Specified_Beh).
equal_states(Module,Nextstate,Derived_State):-
   state_eqn(Module,Nextstate := Function),      /* get specified state */
   eqb(Module,Derived_State,Function).

/*************************************************************/
/*                  TRIVIAL IDENTITY                         */
/*************************************************************/

eqb(M,X,X)  :-!.
```

```
/*************************************************************/
/*                BOOLEAN EXPANSION                          */
/*  The clause eq(NewDB,NewSB) is the driver for the code    */
/*  found in boole2.pro which performs CPT Dukes boolean     */
/*  expansion.                                               */
/*************************************************************/

eqb(M,DB,SB) :-
/* expandable(M),            fewer than too be determined combinations */
                                        /* and boolean variables      */
   evaluate_dukes(DB,NewDB),
   evaluate_dukes(SB,NewSB),
   writeln(['Does ',NewDB,' ='']),writeln(['      ',NewSB]),
   eq(NewDB,NewSB),
   writeln([DB,' ='']),writeln(['     ',SB]),
   writeln(['By Boolean Expansion']),!.

expandable(M) :-                        /* some how_many function will */
   port(M,_,_,boole),!.                 /*  be required                */

/*************************************************************/
/*                      SIMPLIFICATION                       */
/*************************************************************/

eqb(M,DB,SB) :-
   evaluate1(DB,NDB),
   evaluate1(SB,NSB),
   ( DB \== NDB ;
     SB \== NSB ),
   writeln(['Derived behavior is: ',DB]),
   eqb(M,NDB,NSB), !.
```

```
/*************************************************************/
/*                                                           */
/*                        VERIFY.PRO                         */
/*  This clause provides the user interface for the entire   */
/*  VERIFY Prototype. When a user types verify(Module),      */
/*  after loading the appropriate files, the verify clause   */
/*  invokes other clauses to recursively verify each         */
/*  Module.  Each verify clause handles a different type     */
/*  of Module as noted beside each head.  The part and       */
/*  state_eqn clauses provided in Module descriptions are    */
/*  used to determine Module type.  Other clauses used are   */
/*  the following:                                           */
/*                                                           */
/*    derive_and_equate_behaviors:  provides mechanism to    */
/*        derive behavior for each output, and determine     */
/*        equivalence to specified behavior.                 */
/*    derive_and_equate_states:  provides mechanism to       */
/*        derive behavior for each next state, and           */
/*        determine equivalence to specified next state.     */
/*    verify_components:  uses verify to recursively         */
/*        verify components prior to deriving component      */
/*        behavior and next state.                           */
/*************************************************************/

verify(Module) :-                    /*  previously verified module   */
   verified(Module),!,
   writeln(['>>>',Module,' previously verified >>>']).
verify(Module) :-                 /*  primitive module with no state  */
   not part(Module,_,_),
   not state_eqn(Module,_),!,
/*************************************************************/
/*  For a primitive module, behavior = structure.            */
/*  No need to reassert this in the database, since it can   */
/*  be taken from the behavioral specification.              */
/*  output_eqn(Module, Output := Behavior). ALREADY EXISTS   */
/*  If it is decided that derived_behavior should appear as  */
/*  asserta(derived_behavior(Module,Output,Behavior)), the   */
/*  derive_behavior clause dealing with primitives can be    */
/*  removed.  A similar decision is required for the         */
/*  asserta(verified(Module)) for a primitive Module.  With  */
/*  only one possible verified clause per Module, I felt     */
/*  the space required was minimal for the time savings.     */
/*************************************************************/
   asserta(verified(Module)),
   writeln(['>>>',Module,' primitive (needs no verification)>>>']).
```

```
verify(Module) :-                    /*  primitive module  with state  */
   not part(Module,_,_),!,
       /*   Also, no need to reasert next state either.              */
       /*   state_eqn(Module,Nextstate := Function) ALREADY EXISTS   */
   asserta(verified(Module)),
   writeln(['>>>',Module,' primitive (needs no verification)>>>']).
verify(Module) :-                     /* non-primitive with no state  */
   not state_eqn(Module,_),
   writeln(['>>> Attempting to verify ',Module,'>>>']),
   verify_components(Module),!,
/*************************************************************/
/* Derive behavior for all outputs and if equal to           */
/* specified behavior, then assert in database.  This may    */
/* require later garbage collection if the earlier outputs   */
/* are okay, but a later output is not.  This would require  */
/* a cleanup to check the verified(Module) against the       */
/* derived_behavior and next_state clauses. If the clauses   */
/* are not consistent, then remove all derive_behavior and   */
/* next_state clauses.                                       */
/*************************************************************/
   derive_and_equate_behaviors(Module),
   asserta(verified(Module)),
   writeln(['<<<Success! Behavior of',Module,'meets its specification.']).
verify(Module) :-                            /* non-primitive with state  */
   writeln(['>>> Attempting to verify ',Module,'>>>']),
   verify_components(Module),
   !,
   derive_and_equate_behaviors(Module),
   derive_and_equate_states(Module),
   asserta(verified(Module)),
   writeln(['<<<Success! Behavior of',Module,'meets its specification.']).
```

```
/*************************************************************/
/*  The first clause of the next three procedures always     */
/*  succeeds.  We need a way to check that all components    */
/*  are verified, and all behaviors and next_states are      */
/*  equivalent.  The second clause of each procedure does    */
/*  this checking by generating lists of components, states,*/
/*  and outputs and comparing the length of the two lists    */
/*  for the states and outputs since no two state or output */
/*  names should be generated twice for a single module.     */
/*  Setof will generate all Components for a single module, */
/*  so we just check to see if that component was actually   */
/*  verified.                                                */
/*************************************************************/

derive_and_equate_behaviors(Module) :-
   derive_behaviors(Module,Output,Derived_Beh),
   equal_behaviors(Module,Output,Derived_Beh),
   asserta(derived_behavior(Module,Output,Derived_Beh)),
   fail.
derive_and_equate_behaviors(Module) :-
   setof(Outputs,output_eqn(Module,Outputs := _),Outlist),
   length(Outlist,Outnum),
   setof(Outputs,derived_behavior(Module,Outputs,_),Derlist),
   length(Derlist,Dernum),
   Outnum =:= Dernum.
derive_and_equate_behaviors(Module) :-
    retract(derived_behavior(Module,_,_)),
    fail.

derive_and_equate_states(Module) :-
   derive_states(Module,State,Next_State),
   equal_states(Module,State,Next_State),
   asserta(next_state(Module,State,Next_State)),
   fail.
derive_and_equate_states(Module) :-
   setof(States,state_eqn(Module,States := _),Statelist),
   length(Statelist,Statenum),
   setof(States,next_state(Module,States,_),Derlist),
   length(Derlist,Dernum),
   Statenum =:= Dernum.
derive_and_equate_states(Module) :-
    retract(next_state(Module,_,_)),
    fail.
```

```
verify_components(Module) :-
   part(Module,_,Component),
   verify(Component),
   fail.
verify_components(Module) :-
   setof(Component,part(Module,_,Component),Complist),
   parts_verified(Complist),
   writeln(['component list is ', Complist]).

/*************************************************************/
/*                                                           */
/*  parts_verfied - This procedure ensures that all parts    */
/*   (Components) of a Module have been verified.  A list    */
/*   of parts(Components) generated by setof is passed,      */
/*   and parts_verified checks that each one has an asserted*/
/*   verified fact.                                          */
/*                                                           */
/*************************************************************/

parts_verified([]).
parts_verified([Component|Tail]) :-
   verified(Component),
   parts_verified(Tail).
```

```
/*******************************************************/
/*                                                     */
/*  Counter.pro                                        */
/*                                                     */
/*     Module definitions for the counter example      */
/*     in Barrow's VERIFY article.                     */
/*                                                     */
/*******************************************************/

/*----------------- INCREMENTER ----------------------*/

module_name(inc).

port(inc,in(AnInc),input,integer).
port(inc,out(AnInc),output,integer).

/*  Behavior Specification */

output_eqn(inc, out(AnInc) := 1 + in(AnInc)).

/*----------------  MULTIPLEXER ----------------------*/

module_name(mux).

port(mux,in0(AMux),input,integer).
port(mux,in1(AMux),input,integer).
port(mux,switch(AMux),input,boole).
port(mux,out(AMux),output,integer).

/*  Behavior Specification */

output_eqn(mux,  out(AMux) :=  if(switch(AMux),
                                  in1(AMux),
                                  in0(AMux))).
```

```
/*--------------------  REGISTER -------------------*/

module_name(reg).

port(reg,in(AReg),input,integer).
port(reg,out(AReg),output,integer).

/*  Behavior Specification */

state_of(reg,contents(AReg),integer).

output_eqn(reg,out(AReg) := contents(AReg)).

state_eqn(reg,contents(AReg) := in(AReg)).

/*--------------------  COUNTER ----------------------*/

module_name(counter).

port(counter,in(ACounter),input,integer).
port(counter,ctrl(ACounter),input,boole).
port(counter,out(ACounter),output,integer).

part(counter,muxA(ACounter),mux).
part(counter,regA(ACounter),reg).
part(counter,incA(ACounter),inc).

connected(counter,ctrl(ACounter),switch(muxA(ACounter))).
connected(counter,in(ACounter),in1(muxA(ACounter))).
connected(counter,out(muxA(ACounter)),in(regA(ACounter))).
connected(counter,out(regA(ACounter)),in(incA(ACounter))).
connected(counter,out(incA(ACounter)),in0(muxA(ACounter))).
connected(counter,out(regA(ACounter)),out(ACounter)).

/*  Behavior Specification */

state_of(counter,count(ACounter),integer).
state_map(counter,count(ACounter),contents(regA(ACounter))).

output_eqn(counter,out(ACounter) := count(ACounter) ).

state_eqn(counter, count(ACounter) :=  if(ctrl(ACounter),
                                          in(ACounter),
                                          count(ACounter) + 1)).
```

```
/*************************************************************/
/*                        XOR.PRO                            */
/*  This file provides the module descriptions for 2-input   */
/*  nands and exclusive ors.  This file is required when     */
/*  verifying a fulladder.                                   */
/*                                                           */
/*************************************************************/

/*-----  Structural Specification for 2-input NAND ----*/

module_name(nand2).

port(nand2,in0(ANand2),input,boole).
port(nand2,in1(ANand2),input,boole).
port(nand2,out(ANand2),output,boole).

/* Behavioral Specification   */

output_eqn(nand2,
          out(ANand2) := or( neg(in0(ANand2)), neg(in1(ANand2))) ).

/* Structural specification for a two-input Exclusive-OR  */

module_name(xor).

port(xor,in0(AnXor),input,boole).
port(xor,in1(AnXor),input,boole).
port(xor,out(AnXor),output,boole).

part(xor,g1(AnXor),nand2).
part(xor,g2(AnXor),nand2).
part(xor,g3(AnXor),nand2).
part(xor,g4(AnXor),nand2).

connected(xor,in0(AnXor),in0(g1(AnXor))).
connected(xor,in1(AnXor),in1(g1(AnXor))).
connected(xor,in0(AnXor),in0(g2(AnXor))).
connected(xor,out(g1(AnXor)),in1(g2(AnXor))).
connected(xor,out(g1(AnXor)),in0(g3(AnXor))).
connected(xor,in1(AnXor),in1(g3(AnXor))).
connected(xor,out(g2(AnXor)),in0(g4(AnXor))).
connected(xor,out(g3(AnXor)),in1(g4(AnXor))).
connected(xor,out(g4(AnXor)),out(AnXor)).
```

```
/* Behavioral Specification for a two-input XOR */

output_eqn(xor,
          out(AnXor) :=  or( and( neg(in0(AnXor)),
                                  in1(AnXor) ),
                             and( in0(AnXor),
                                  neg(in1(AnXor)) ))).
```

```
/*************************************************************/
/*                      FADDXOR.PRO                          */
/*  This file provides the specification of a fulladder      */
/*  composed of nand and exclusive or gates.  The file       */
/*  xor.pro must also be loaded to provide the module        */
/*  specifications for nand2 and xor.                        */
/*                                                           */
/*************************************************************/

/* Structural specification for a full adder with xors   */

module_name(faddxor).

port(faddxor,x(Afaddxor),input,boole).
port(faddxor,y(Afaddxor),input,boole).
port(faddxor,cin(Afaddxor),input,boole).
port(faddxor,outcarry(Afaddxor),output,boole).
port(faddxor,outsum(Afaddxor),output,boole).

part(faddxor,g1(Afaddxor),nand2).
part(faddxor,g2(Afaddxor),nand2).
part(faddxor,g3(Afaddxor),nand2).
part(faddxor,g4(Afaddxor),xor).
part(faddxor,g5(Afaddxor),xor).

connected(faddxor,x(Afaddxor),in0(g1(Afaddxor))).
connected(faddxor,y(Afaddxor),in1(g1(Afaddxor))).
connected(faddxor,cin(Afaddxor),in0(g2(Afaddxor))).
connected(faddxor,out(g4(Afaddxor)),in1(g2(Afaddxor))).
connected(faddxor,out(g1(Afaddxor)),in0(g3(Afaddxor))).
connected(faddxor,out(g2(Afaddxor)),in1(g3(Afaddxor))).
connected(faddxor,x(Afaddxor),in0(g4(Afaddxor))).
connected(faddxor,y(Afaddxor),in1(g4(Afaddxor))).
connected(faddxor,out(g4(Afaddxor)),in0(g5(Afaddxor))).
connected(faddxor,cin(Afaddxor),in1(g5(Afaddxor))).
connected(faddxor,out(g5(Afaddxor)),outsum(Afaddxor)).
connected(faddxor,out(g3(Afaddxor)),outcarry(Afaddxor)).
```

```
/* Behavioral Specification          */

output_eqn(faddxor,
           outcarry(Afaddxor) :=
             or( and( x(Afaddxor),y(Afaddxor)),
                 and( cin(Afaddxor),
                      xor( x(Afaddxor),y(Afaddxor))  ))).
output_eqn(faddxor,
           outsum(Afaddxor) :=
            xor( xor(x(Afaddxor),y(Afaddxor)),
                 cin(Afaddxor) )).
```

Appendix B. Sample Sessions

This Appendix includes sample runs of a counter-circuit and fulladder on Prolog-1 and Quintus Prolog.

```
/*********************************************************/
/*  This file loads the appropriate files to verify a    */
/*  a counter with Quintus Prolog.  This is invoked by   */
/*  typing ['qctrld.pro'].                               */
/*********************************************************/

?- ['qops.pro','eval.pro','derbeh.pro','derstate.pro','counter.pro'].
?- ['boole2.pro','eqbeh.pro','verify.pro'].

        Sample Run for a counter-circuit on Quintus Prolog:

[1]cub prolog
Quintus Prolog Release 2.4 (VAX, Ultrix 2.0-2.2)
Copyright (C) 1988, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- ['qctrld.pro'].
[consulting /usr/users/gce90d/ksparks/15jan91/qctrld.pro...]
 [consulting /usr/users/gce90d/ksparks/15jan91/qops.pro...]
[Undefined procedures will just fail ('fail' option)]
 [qops.pro consulted 0.367 sec 1,092 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/eval.pro...]
[WARNING: Singleton variables, clause 3 of evaluate_brown/2: F]
 [eval.pro consulted 1.050 sec 2,920 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/derbeh.pro...]
[WARNING: Singleton variables, clause 1 of derive_behaviors/3: Spec_Behavior]
[WARNING: Singleton variables, clause 2 of derive_behaviors/3: Spec_Behavior]
[WARNING: Clauses for derive_behavior/3 are not together in the source file]
[WARNING: Singleton variables, clause 1 of derive_behavior/3: F]
[WARNING: Singleton variables, clause 2 of derive_behavior/3: F]
[WARNING: Singleton variables, clause 8 of derive_behavior/3: Module]
 [derbeh.pro consulted 1.466 sec 3,220 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/derstate.pro...]
[WARNING: Singleton variables, clause 1 of derive_states/3: Type, Part]
[WARNING: Singleton variables, clause 1 of replace_all/5: Part]
[WARNING: Singleton variables, clause 2 of replace_all/5: Module, Old, New]
[WARNING: Singleton variables, clause 1 of replace/4: Old, New]
```

```
[WARNING: Singleton variables, clause 2 of replace/4: Old, New]
[WARNING: Singleton variables, clause 3 of replace/4: New, Arg1]
[WARNING: Singleton variables, clause 11 of replace/4: Old, New]
 [derstate.pro consulted 1.384 sec 2,872 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/counter.pro...]
[WARNING: Singleton variables, clause 1 of port/4: AnInc]
[WARNING: Singleton variables, clause 2 of port/4: AnInc]
[WARNING: Clauses for module_name/1 are not together in the source file]
[WARNING: Clauses for port/4 are not together in the source file]
[WARNING: Singleton variables, clause 1 of port/4: AMux]
[WARNING: Singleton variables, clause 2 of port/4: AMux]
[WARNING: Singleton variables, clause 3 of port/4: AMux]
[WARNING: Singleton variables, clause 4 of port/4: AMux]
[WARNING: Clauses for output_eqn/2 are not together in the source file]
[WARNING: Singleton variables, clause 1 of port/4: AReg]
[WARNING: Singleton variables, clause 2 of port/4: AReg]
[WARNING: Singleton variables, clause 1 of state_of/3: AReg]
[WARNING: Singleton variables, clause 1 of port/4: ACounter]
[WARNING: Singleton variables, clause 2 of port/4: ACounter]
[WARNING: Singleton variables, clause 3 of port/4: ACounter]
[WARNING: Singleton variables, clause 1 of part/3: ACounter]
[WARNING: Singleton variables, clause 2 of part/3: ACounter]
[WARNING: Singleton variables, clause 3 of part/3: ACounter]
[WARNING: Clauses for state_of/3 are not together in the source file]
[WARNING: Singleton variables, clause 1 of state_of/3: ACounter]
[WARNING: Clauses for state_eqn/2 are not together in the source file]
 [counter.pro consulted 1.766 sec 3,804 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/boole2.pro...]
[WARNING: Singleton variables, clause 1 of remove_x_1/3: X]
[WARNING: Singleton variables, clause 2 of remove_x_1/3: X]
[WARNING: Singleton variables, clause 3 of remove_x_1/3: Arg2]
[WARNING: Singleton variables, clause 4 of remove_x_1/3: Arg, Arg1, Arg2]
[WARNING: Singleton variables, clause 5 of remove_x_1/3: Arg2]
[WARNING: Singleton variables, clause 6 of remove_x_1/3: Arg2]
[WARNING: Singleton variables, clause 1 of remove_x_0/3: X]
[WARNING: Singleton variables, clause 2 of remove_x_0/3: X]
[WARNING: Singleton variables, clause 3 of remove_x_0/3: Arg2]
[WARNING: Singleton variables, clause 4 of remove_x_0/3: Arg2]
[WARNING: Singleton variables, clause 5 of remove_x_0/3: Arg2]
[WARNING: Singleton variables, clause 6 of remove_x_0/3: Arg2]
 [boole2.pro consulted 2.600 sec 6,820 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/eqbeh.pro...]
[WARNING: Singleton variables, clause 1 of eqb/3: M]
[WARNING: Singleton variables, clause 2 of eqb/3: M]
[WARNING: Clauses for eqb/3 are not together in the source file]
```

```
 [eqbeh.pro consulted 0.600 sec 1,244 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/verify.pro...]
 [verify.pro consulted 1.134 sec 3,036 bytes]
[qctrld.pro consulted 11.166 sec 25,776 bytes]
yes
| ?- verify(counter).
>>> Attempting to verify counter>>>
>>>mux primitive (needs no verification)>>>
>>>reg primitive (needs no verification)>>>
>>>inc primitive (needs no verification)>>>
Applying Rule 1B to out(_681)
Applying Rule 2A to out(regA(_681))
reg's output equation:
   out(regA(_681)) := contents(regA(_681))
Applying default Rule to contents(regA(_681))
Derived Behavior:  contents(regA(_681))
Rule 1
Rule4
Rule4
Rule4
Value of count(_681):
          count(_681)
Value of count(_681):
          count(_681)
Value of count(_681):
          count(_681)
Substituted Behavior:  count(_681)
Applying Rule 1B to in(regA(_1274))
Applying Rule 2A to out(muxA(_1274))
mux's output equation:
   out(muxA(_1274)) := if(switch(muxA(_1274)),in1(muxA(_1274)),in0(muxA(_1274)))
Applying Rule 6 to if(switch(muxA(_1274)),in1(muxA(_1274)),in0(muxA(_1274)))
Applying Rule 1A to switch(muxA(_1274))
Applying Rule 1A to in1(muxA(_1274))
Applying Rule 1B to in0(muxA(_1274))
Applying Rule 2A to out(incA(_1274))
inc's output equation:
   out(incA(_1274)) := 1+in(incA(_1274))
Applying Rule 7 to 1+in(incA(_1274))
Applying default Rule to 1
Applying Rule 1B to in(incA(_1274))
Applying Rule 2A to out(regA(_1274))
reg's output equation:
   out(regA(_1274)) := contents(regA(_1274))
Applying default Rule to contents(regA(_1274))
```

```
Value of 1+contents(regA(_1274)):
         1+contents(regA(_1274))
Value of if(ctrl(_1274),in(_1274),1+contents(regA(_1274))):
         if(ctrl(_1274),in(_1274),1+contents(regA(_1274)))
Derived Behavior:  if(ctrl(_1274),in(_1274),1+contents(regA(_1274)))
Rule if
Rule4
Rule4
Rule +
Rule2
Rule 1
Rule if
Rule4
Rule4
Rule +
Rule2
Rule4
Rule if
Rule4
Rule3
Rule struct
Rule +
Rule2
Rule4
Rule if
Rule4
Rule4
Rule +
Rule2
Rule4
Value of if(ctrl(_1274),in(_1274),1+count(_1274)):
         if(ctrl(_1274),in(_1274),1+count(_1274))
Value of if(ctrl(_1274),in(_1274),1+count(_1274)):
         if(ctrl(_1274),in(_1274),1+count(_1274))
Value of if(ctrl(_1274),in(_1274),1+count(_1274)):
         if(ctrl(_1274),in(_1274),1+count(_1274))
Substituted Behavior:  if(ctrl(_1274),in(_1274),1+count(_1274))
Does if(ctrl(_1274),in(_1274),count(_1274)+1) =
     if(ctrl(_1274),in(_1274),1+count(_1274))
Value of if(ctrl(_1274),in(_1274),count(_1274)+1):
         if(ctrl(_1274),in(_1274),1+count(_1274))
Derived behavior is: if(ctrl(_1274),in(_1274),count(_1274)+1)
<<< Success! Behavior of counter meets its specification.
yes
| ?- halt.
```

```
/*************************************************************/
/*  This file loads the appropriate files to verify a        */
/*  a counter on PROLOG-1.  This is invoked by typing        */
/*     ['a:ctrload'].                                        */
/*************************************************************/

?- ['a:ops','a:eval','a:derbeh','a:derstate','a:counter'].
?- ['a:boole2','a:eqbeh','a:verify'].

     Sample run for a counter-circuit on PROLOG-1:

B:\>prolog

+-----------------------------------------+
| MS-DOS Prolog-1             Version 2.2 |
| Copyright 1983      Serial number: 0001213 |
| Expert Systems Ltd.                     |
| Oxford U.K.                             |
+-----------------------------------------+

?- ['a:ctrload'].
a:ops consulted.
a:eval consulted.
a:derbeh consulted.
a:derstate consulted.
a:counter consulted.
a:boole2 consulted.
a:eqbeh consulted.
a:verify consulted.
a:ctrload consulted.
?- verify(counter).
>>> Attempting to verify counter>>>
>>>mux primitive (needs no verification)>>>
>>>reg primitive (needs no verification)>>>
>>>inc primitive (needs no verification)>>>
Applying Rule 1B to out(_99)
Applying Rule 2A to out(regA(_99))
reg's output equation:
   out(regA(_99)) := contents(regA(_99))
Applying default Rule to contents(regA(_99))
Derived Behavior:  contents(regA(_99))
Rule 1
Rule4
Rule4
Rule4
```

```
Value of count(_99):
         count(_99)
Value of count(_99):
         count(_99)
Value of count(_99):
         count(_99)
Substituted Behavior:  count(_99)
Applying Rule 1B to in(regA(_171))
Applying Rule 2A to out(muxA(_171))
mux's output equation:
   out(muxA(_171)) := if(switch(muxA(_171)),in1(muxA(_171)),in0(muxA(_171)))
Applying Rule 6 to if(switch(muxA(_171)),in1(muxA(_171)),in0(muxA(_171)))
Applying Rule 1A to switch(muxA(_171))
Applying Rule 1A to in1(muxA(_171))
Applying Rule 1B to in0(muxA(_171))
Applying Rule 2A to out(incA(_171))
inc's output equation:
   out(incA(_171)) := 1+in(incA(_171))
Applying Rule 7 to 1+in(incA(_171))
Applying default Rule to 1
Applying Rule 1B to in(incA(_171))
Applying Rule 2A to out(regA(_171))
reg's output equation:
   out(regA(_171)) := contents(regA(_171))
Applying default Rule to contents(regA(_171))
Value of 1+contents(regA(_171)):
         1+contents(regA(_171))
Value of if(ctrl(_171),in(_171),1+contents(regA(_171))):
         if(ctrl(_171),in(_171),1+contents(regA(_171)))
Derived Behavior:  if(ctrl(_171),in(_171),1+contents(regA(_171)))
Rule if
Rule4
Rule4
Rule +
Rule2
Rule 1
Rule if
Rule4
Rule4
Rule +
Rule2
Rule4
Rule if
Rule4
Rule3
```

```
Rule struct
Rule +
Rule2
Rule4
Rule if
Rule4
Rule4
Rule +
Rule2
Rule4
Value of if(ctrl(_171),in(_171),1+count(_171)):
         if(ctrl(_171),in(_171),1+count(_171))
Value of if(ctrl(_171),in(_171),1+count(_171)):
         if(ctrl(_171),in(_171),1+count(_171))
Value of if(ctrl(_171),in(_171),1+count(_171)):
         if(ctrl(_171),in(_171),1+count(_171))
Substituted Behavior:  if(ctrl(_171),in(_171),1+count(_171))
Does if(ctrl(_171),in(_171),count(_171)+1) =
     if(ctrl(_171),in(_171),1+count(_171))
Value of if(ctrl(_171),in(_171),count(_171)+1):
         if(ctrl(_171),in(_171),1+count(_171))
Derived behavior is: if(ctrl(_171),in(_171),count(_171)+1)
<<< Success! Behavior of counter meets its specification.

yes
?- halt.
```

```
/*************************************************************/
/*  This file loads the appropriate files to verify a        */
/*  a fulladder with Quintus Prolog.  This is invoked by     */
/*  typing ['qfaddld.pro'].                                  */
/*************************************************************/

?- ['qops.pro','eval.pro','derbeh.pro'].
?- ['derstate.pro','xor.pro','faddxor.pro'].
?- ['boole2.pro','eqbeh.pro','verify.pro'].

     Sample run for a full-adder circuit on Quintus Prolog:

[1]cub prolog

Quintus Prolog Release 2.4 (VAX, Ultrix 2.0-2.2)
Copyright (C) 1988, Quintus Computer Systems, Inc.  All rights reserved.
1310 Villa Street, Mountain View, California  (415) 965-7700

| ?- ['qfaddld.pro'].
[consulting /usr/users/gce90d/ksparks/15jan91/qfaddld.pro...]
 [consulting /usr/users/gce90d/ksparks/15jan91/qops.pro...]
[Undefined procedures will just fail ('fail' option)]
 [qops.pro consulted 0.383 sec 1,092 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/eval.pro...]
[WARNING: Singleton variables, clause 3 of evaluate_brown/2: F]
 [eval.pro consulted 1.050 sec 2,936 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/derbeh.pro...]
[WARNING: Singleton variables, clause 1 of derive_behaviors/3: Spec_Behavior]
[WARNING: Singleton variables, clause 2 of derive_behaviors/3: Spec_Behavior]
[WARNING: Clauses for derive_behavior/3 are not together in the source file]
[WARNING: Singleton variables, clause 1 of derive_behavior/3: F]
[WARNING: Singleton variables, clause 2 of derive_behavior/3: F]
[WARNING: Singleton variables, clause 8 of derive_behavior/3: Module]
 [derbeh.pro consulted 1.450 sec 3,220 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/derstate.pro...]
[WARNING: Singleton variables, clause 1 of derive_states/3: Type, Part]
[WARNING: Singleton variables, clause 1 of replace_all/5: Part]
[WARNING: Singleton variables, clause 2 of replace_all/5: Module, Old, New]
[WARNING: Singleton variables, clause 1 of replace/4: Old, New]
[WARNING: Singleton variables, clause 2 of replace/4: Old, New]
[WARNING: Singleton variables, clause 3 of replace/4: New, Arg1]
[WARNING: Singleton variables, clause 11 of replace/4: Old, New]
 [derstate.pro consulted 1.383 sec 2,904 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/xor.pro...]
[WARNING: Singleton variables, clause 1 of port/4: ANand2]
```

```
[WARNING: Singleton variables, clause 2 of port/4: ANand2]
[WARNING: Singleton variables, clause 3 of port/4: ANand2]
[WARNING: Clauses for module_name/1 are not together in the source file]
[WARNING: Clauses for port/4 are not together in the source file]
[WARNING: Singleton variables, clause 1 of port/4: AnXor]
[WARNING: Singleton variables, clause 2 of port/4: AnXor]
[WARNING: Singleton variables, clause 3 of port/4: AnXor]
[WARNING: Singleton variables, clause 1 of part/3: AnXor]
[WARNING: Singleton variables, clause 2 of part/3: AnXor]
[WARNING: Singleton variables, clause 3 of part/3: AnXor]
[WARNING: Singleton variables, clause 4 of part/3: AnXor]
[WARNING: Clauses for output_eqn/2 are not together in the source file]
 [xor.pro consulted 1.184 sec 2,616 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/faddxor.pro...]
[WARNING: Singleton variables, clause 1 of port/4: Afaddxor]
[WARNING: Singleton variables, clause 2 of port/4: Afaddxor]
[WARNING: Singleton variables, clause 3 of port/4: Afaddxor]
[WARNING: Singleton variables, clause 4 of port/4: Afaddxor]
[WARNING: Singleton variables, clause 5 of port/4: Afaddxor]
[WARNING: Singleton variables, clause 1 of part/3: Afaddxor]
[WARNING: Singleton variables, clause 2 of part/3: Afaddxor]
[WARNING: Singleton variables, clause 3 of part/3: Afaddxor]
[WARNING: Singleton variables, clause 4 of part/3: Afaddxor]
[WARNING: Singleton variables, clause 5 of part/3: Afaddxor]
 [faddxor.pro consulted 1.183 sec 2,368 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/boole2.pro...]
[WARNING: Singleton variables, clause 1 of remove_x_1/3: X]
[WARNING: Singleton variables, clause 2 of remove_x_1/3: X]
[WARNING: Singleton variables, clause 3 of remove_x_1/3: Arg2]
[WARNING: Singleton variables, clause 4 of remove_x_1/3: Arg, Arg1, Arg2]
[WARNING: Singleton variables, clause 5 of remove_x_1/3: Arg2]
[WARNING: Singleton variables, clause 6 of remove_x_1/3: Arg2]
[WARNING: Singleton variables, clause 1 of remove_x_0/3: X]
[WARNING: Singleton variables, clause 2 of remove_x_0/3: X]
[WARNING: Singleton variables, clause 3 of remove_x_0/3: Arg2]
[WARNING: Singleton variables, clause 4 of remove_x_0/3: Arg2]
[WARNING: Singleton variables, clause 5 of remove_x_0/3: Arg2]
[WARNING: Singleton variables, clause 6 of remove_x_0/3: Arg2]
 [boole2.pro consulted 2.583 sec 6,900 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/eqbeh.pro...]
[WARNING: Singleton variables, clause 1 of eqb/3: M]
[WARNING: Singleton variables, clause 2 of eqb/3: M]
[WARNING: Clauses for eqb/3 are not together in the source file]
 [eqbeh.pro consulted 0.600 sec 1,268 bytes]
 [consulting /usr/users/gce90d/ksparks/15jan91/verify.pro...]
```

```
 [verify.pro consulted 1.100 sec 3,004 bytes]
[qfaddld.pro consulted 11.800 sec 27,116 bytes]
yes
| ?- verify(faddxor).
>>> Attempting to verify faddxor>>>
>>>nand2 primitive (needs no verification)>>>
>>>nand2 previously verified >>>
>>>nand2 previously verified >>>
>>> Attempting to verify xor>>>
>>>nand2 previously verified >>>
>>>nand2 previously verified >>>
>>>nand2 previously verified >>>
>>>nand2 previously verified >>>
Applying Rule 1B to out(_1068)
Applying Rule 2A to out(g4(_1068))
nand2's output equation:
   out(g4(_1068)) := or(neg(in0(g4(_1068))),neg(in1(g4(_1068))))
Applying Rule 5 to or(neg(in0(g4(_1068))),neg(in1(g4(_1068))))
Applying Rule 3 to neg(in0(g4(_1068)))
Applying Rule 1B to in0(g4(_1068))
Applying Rule 2A to out(g2(_1068))
nand2's output equation:
   out(g2(_1068)) := or(neg(in0(g2(_1068))),neg(in1(g2(_1068))))
Applying Rule 5 to or(neg(in0(g2(_1068))),neg(in1(g2(_1068))))
Applying Rule 3 to neg(in0(g2(_1068)))
Applying Rule 1A to in0(g2(_1068))
Value of neg(in0(_1068)):
         neg(in0(_1068))
Applying Rule 3 to neg(in1(g2(_1068)))
Applying Rule 1B to in1(g2(_1068))
Applying Rule 2A to out(g1(_1068))
nand2's output equation:
   out(g1(_1068)) := or(neg(in0(g1(_1068))),neg(in1(g1(_1068))))
Applying Rule 5 to or(neg(in0(g1(_1068))),neg(in1(g1(_1068))))
Applying Rule 3 to neg(in0(g1(_1068)))
Applying Rule 1A to in0(g1(_1068))
Value of neg(in0(_1068)):
         neg(in0(_1068))
Applying Rule 3 to neg(in1(g1(_1068)))
Applying Rule 1A to in1(g1(_1068))
Value of neg(in1(_1068)):
         neg(in1(_1068))
Value of or(neg(in0(_1068)),neg(in1(_1068))):
         or(neg(in0(_1068)),neg(in1(_1068)))
Value of neg(or(neg(in0(_1068)),neg(in1(_1068)))):
```

```
          and(in0(_1068),in1(_1068))
Value of or(neg(in0(_1068)),and(in0(_1068),in1(_1068))):
          or(neg(in0(_1068)),and(in0(_1068),in1(_1068)))
Value of neg(or(neg(in0(_1068)),and(in0(_1068),in1(_1068)))):
          and(in0(_1068),or(neg(in0(_1068)),neg(in1(_1068))))
Applying Rule 3 to neg(in1(g4(_1068)))
Applying Rule 1B to in1(g4(_1068))
Applying Rule 2A to out(g3(_1068))
nand2's output equation:
   out(g3(_1068)) := or(neg(in0(g3(_1068))),neg(in1(g3(_1068))))
Applying Rule 5 to or(neg(in0(g3(_1068))),neg(in1(g3(_1068))))
Applying Rule 3 to neg(in0(g3(_1068)))
Applying Rule 1B to in0(g3(_1068))
Applying Rule 2A to out(g1(_1068))
nand2's output equation:
   out(g1(_1068)) := or(neg(in0(g1(_1068))),neg(in1(g1(_1068))))
Applying Rule 5 to or(neg(in0(g1(_1068))),neg(in1(g1(_1068))))
Applying Rule 3 to neg(in0(g1(_1068)))
Applying Rule 1A to in0(g1(_1068))
Value of neg(in0(_1068)):
          neg(in0(_1068))
Applying Rule 3 to neg(in1(g1(_1068)))
Applying Rule 1A to in1(g1(_1068))
Value of neg(in1(_1068)):
          neg(in1(_1068))
Value of or(neg(in0(_1068)),neg(in1(_1068))):
          or(neg(in0(_1068)),neg(in1(_1068)))
Value of neg(or(neg(in0(_1068)),neg(in1(_1068)))):
          and(in0(_1068),in1(_1068))
Applying Rule 3 to neg(in1(g3(_1068)))
Applying Rule 1A to in1(g3(_1068))
Value of neg(in1(_1068)):
          neg(in1(_1068))
Value of or(and(in0(_1068),in1(_1068)),neg(in1(_1068))):
          or(and(in0(_1068),in1(_1068)),neg(in1(_1068)))
Value of neg(or(and(in0(_1068),in1(_1068)),neg(in1(_1068)))):
          and(or(neg(in0(_1068)),neg(in1(_1068))),in1(_1068))
Value of or(and(in0(_1068),or(neg(in0(_1068)),neg(in1(_1068)))),
            and(or(neg(in0(_1068)),neg(in1(_1068))),in1(_1068))):
         or(and(in0(_1068),or(neg(in0(_1068)),neg(in1(_1068)))),
            and(or(neg(in0(_1068)),neg(in1(_1068))),in1(_1068)))
Does or(and(in0(_1068),or(neg(in0(_1068)),neg(in1(_1068)))),
       and(or(neg(in0(_1068)),neg(in1(_1068))),in1(_1068))) =
    or(and(neg(in0(_1068)),in1(_1068)),and(in0(_1068),neg(in1(_1068))))
or(and(in0(_1068),or(neg(in0(_1068)),neg(in1(_1068)))),
```

```
   and(or(neg(in0(_1068)),neg(in1(_1068))),in1(_1068))) =
or(and(neg(in0(_1068)),in1(_1068)),and(in0(_1068),neg(in1(_1068))))
By Boolean Expansion
<<< Success! Behavior of xor meets its specification.
>>>xor previously verified >>>
component list is [nand2]
Applying Rule 1B to outcarry(_804)
Applying Rule 2A to out(g3(_804))
nand2's output equation:
   out(g3(_804)) := or(neg(in0(g3(_804))),neg(in1(g3(_804))))
Applying Rule 5 to or(neg(in0(g3(_804))),neg(in1(g3(_804))))
Applying Rule 3 to neg(in0(g3(_804)))
Applying Rule 1B to in0(g3(_804))
Applying Rule 2A to out(g1(_804))
nand2's output equation:
   out(g1(_804)) := or(neg(in0(g1(_804))),neg(in1(g1(_804))))
Applying Rule 5 to or(neg(in0(g1(_804))),neg(in1(g1(_804))))
Applying Rule 3 to neg(in0(g1(_804)))
Applying Rule 1A to in0(g1(_804))
Value of neg(x(_804)):
         neg(x(_804))
Applying Rule 3 to neg(in1(g1(_804)))
Applying Rule 1A to in1(g1(_804))
Value of neg(y(_804)):
         neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
         or(neg(x(_804)),neg(y(_804)))
Value of neg(or(neg(x(_804)),neg(y(_804)))):
         and(x(_804),y(_804))
Applying Rule 3 to neg(in1(g3(_804)))
Applying Rule 1B to in1(g3(_804))
Applying Rule 2A to out(g2(_804))
nand2's output equation:
   out(g2(_804)) := or(neg(in0(g2(_804))),neg(in1(g2(_804))))
Applying Rule 5 to or(neg(in0(g2(_804))),neg(in1(g2(_804))))
Applying Rule 3 to neg(in0(g2(_804)))
Applying Rule 1A to in0(g2(_804))
Value of neg(cin(_804)):
         neg(cin(_804))
Applying Rule 3 to neg(in1(g2(_804)))
Applying Rule 1B to in1(g2(_804))
Applying Rule 2B to out(g4(_804))
xor's derived behavior:
   out(g4(_804)) := or(and(in0(g4(_804)),
                           or(neg(in0(g4(_804))),neg(in1(g4(_804))))),
```

```
                    and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
                        in1(g4(_804))))
Applying Rule 5 to or(and(in0(g4(_804)),
                       or(neg(in0(g4(_804))),neg(in1(g4(_804))))),
                    and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
                       in1(g4(_804))))
Applying Rule 4 to and(in0(g4(_804)),
                     or(neg(in0(g4(_804))),neg(in1(g4(_804)))))
Applying Rule 1A to in0(g4(_804))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
         neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
         neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
         or(neg(x(_804)),neg(y(_804)))
Value of and(x(_804),or(neg(x(_804)),neg(y(_804)))):
         and(x(_804),or(neg(x(_804)),neg(y(_804))))
Applying Rule 4 to and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
                     in1(g4(_804)))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
         neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
         neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
         or(neg(x(_804)),neg(y(_804)))
Applying Rule 1A to in1(g4(_804))
Value of and(or(neg(x(_804)),neg(y(_804))),y(_804)):
        and(or(neg(x(_804)),neg(y(_804))),y(_804))
Value of or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
           and(or(neg(x(_804)),neg(y(_804))),y(_804))):
        or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
           and(or(neg(x(_804)),neg(y(_804))),y(_804)))
Value of neg(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
                and(or(neg(x(_804)),neg(y(_804))),y(_804)))):
        and(or(neg(x(_804)),and(x(_804),y(_804))),
```

```
            or(and(x(_804),y(_804)),neg(y(_804))))
Value of or(neg(cin(_804)),
           and(or(neg(x(_804)),and(x(_804),y(_804))),
               or(and(x(_804),y(_804)),neg(y(_804))))):
        or(neg(cin(_804)),
           and(or(neg(x(_804)),and(x(_804),y(_804))),
               or(and(x(_804),y(_804)),neg(y(_804)))))
Value of neg(or(neg(cin(_804)),
                and(or(neg(x(_804)),and(x(_804),y(_804))),
                    or(and(x(_804),y(_804)),neg(y(_804))))))):
        and(cin(_804),or(and(x(_804),
                             or(neg(x(_804)),neg(y(_804)))),
                         and(or(neg(x(_804)),neg(y(_804))),
                             y(_804))))
Value of or(and(x(_804),y(_804)),
           and(cin(_804),
               or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
                  and(or(neg(x(_804)),neg(y(_804))),y(_804))))):
        or(and(x(_804),y(_804)),
           and(cin(_804),
               or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
                  and(or(neg(x(_804)),neg(y(_804))),y(_804)))))
Does or(and(x(_804),y(_804)),
        and(cin(_804),
            or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804))))) =
     or(and(x(_804),y(_804)),and(cin(_804),xor(x(_804),y(_804))))
or(and(x(_804),y(_804)),
   and(cin(_804),
       or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
          and(or(neg(x(_804)),neg(y(_804))),y(_804))))) =
or(and(x(_804),y(_804)),and(cin(_804),xor(x(_804),y(_804))))
By Boolean Expansion
Applying Rule 1B to outsum(_804)
Applying Rule 2B to out(g5(_804))
xor's derived behavior:
   out(g5(_804)) := or(and(in0(g5(_804)),
                           or(neg(in0(g5(_804))),neg(in1(g5(_804))))),
                       and(or(neg(in0(g5(_804))),neg(in1(g5(_804)))),
                           in1(g5(_804))))
Applying Rule 5 to or(and(in0(g5(_804)),
                          or(neg(in0(g5(_804))),neg(in1(g5(_804))))),
                      and(or(neg(in0(g5(_804))),neg(in1(g5(_804)))),
                          in1(g5(_804))))
Applying Rule 4 to and(in0(g5(_804)),
```

```
                    or(neg(in0(g5(_804))),neg(in1(g5(_804)))))
Applying Rule 1B to in0(g5(_804))
Applying Rule 2B to out(g4(_804))
xor's derived behavior:
   out(g4(_804)) := or(and(in0(g4(_804)),
                           or(neg(in0(g4(_804))),neg(in1(g4(_804))))),
                       and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
                           in1(g4(_804))))
Applying Rule 5 to or(and(in0(g4(_804)),
                          or(neg(in0(g4(_804))),neg(in1(g4(_804))))),
                      and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
                          in1(g4(_804))))
Applying Rule 4 to and(in0(g4(_804)),
                       or(neg(in0(g4(_804))),neg(in1(g4(_804)))))
Applying Rule 1A to in0(g4(_804))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
         neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
         neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
         or(neg(x(_804)),neg(y(_804)))
Value of and(x(_804),or(neg(x(_804)),neg(y(_804)))):
         and(x(_804),or(neg(x(_804)),neg(y(_804))))
Applying Rule 4 to and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
                       in1(g4(_804)))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
         neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
         neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
         or(neg(x(_804)),neg(y(_804)))
Applying Rule 1A to in1(g4(_804))
Value of and(or(neg(x(_804)),neg(y(_804))),y(_804)):
         and(or(neg(x(_804)),neg(y(_804))),y(_804))
Value of or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
```

```
          and(or(neg(x(_804)),neg(y(_804))),y(_804))):
        or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
          and(or(neg(x(_804)),neg(y(_804))),y(_804)))
Applying Rule 5 to or(neg(in0(g5(_804))),neg(in1(g5(_804))))
Applying Rule 3 to neg(in0(g5(_804)))
Applying Rule 1B to in0(g5(_804))
Applying Rule 2B to out(g4(_804))
xor's derived behavior:
   out(g4(_804)) := or(and(in0(g4(_804)),
                           or(neg(in0(g4(_804))),neg(in1(g4(_804))))),
                       and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
                           in1(g4(_804))))
Applying Rule 5 to or(and(in0(g4(_804)),
                          or(neg(in0(g4(_804))),neg(in1(g4(_804))))),
                      and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
                          in1(g4(_804))))
Applying Rule 4 to and(in0(g4(_804)),
                       or(neg(in0(g4(_804))),neg(in1(g4(_804)))))
Applying Rule 1A to in0(g4(_804))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
          neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
          neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
          or(neg(x(_804)),neg(y(_804)))
Value of and(x(_804),or(neg(x(_804)),neg(y(_804)))):
          and(x(_804),or(neg(x(_804)),neg(y(_804))))
Applying Rule 4 to and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),in1(g4(_804)))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
          neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
          neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
          or(neg(x(_804)),neg(y(_804)))
Applying Rule 1A to in1(g4(_804))
```

```
Value of and(or(neg(x(_804)),neg(y(_804))),y(_804)):
          and(or(neg(x(_804)),neg(y(_804))),y(_804))
Value of or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
            and(or(neg(x(_804)),neg(y(_804))),y(_804))):
         or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
            and(or(neg(x(_804)),neg(y(_804))),y(_804)))
Value of neg(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
                and(or(neg(x(_804)),neg(y(_804))),y(_804)))):
         and(or(neg(x(_804)),and(x(_804),y(_804))),
             or(and(x(_804),y(_804)),neg(y(_804))))
Applying Rule 3 to neg(in1(g5(_804)))
Applying Rule 1A to in1(g5(_804))
Value of neg(cin(_804)):
          neg(cin(_804))
Value of or(and(or(neg(x(_804)),and(x(_804),y(_804))),
                or(and(x(_804),y(_804)),neg(y(_804)))),
            neg(cin(_804))):
         or(and(or(neg(x(_804)),and(x(_804),y(_804))),
                or(and(x(_804),y(_804)),neg(y(_804)))),
            neg(cin(_804)))
Value of and(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
                and(or(neg(x(_804)),neg(y(_804))),y(_804))),
             or(and(or(neg(x(_804)),and(x(_804),y(_804))),
                    or(and(x(_804),y(_804)),neg(y(_804)))),
                neg(cin(_804)))):
         and(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
                and(or(neg(x(_804)),neg(y(_804))),y(_804))),
             or(and(or(neg(x(_804)),and(x(_804),y(_804))),
                    or(and(x(_804),y(_804)),neg(y(_804)))),
                neg(cin(_804))))
Applying Rule 4 to and(or(neg(in0(g5(_804))),neg(in1(g5(_804)))),
                       in1(g5(_804)))
Applying Rule 5 to or(neg(in0(g5(_804))),neg(in1(g5(_804))))
Applying Rule 3 to neg(in0(g5(_804)))
Applying Rule 1B to in0(g5(_804))
Applying Rule 2B to out(g4(_804))
xor's derived behavior:
   out(g4(_804)) := or(and(in0(g4(_804)),
                           or(neg(in0(g4(_804))),neg(in1(g4(_804))))),
                       and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
                           in1(g4(_804))))
Applying Rule 5 to or(and(in0(g4(_804)),
                          or(neg(in0(g4(_804))),neg(in1(g4(_804))))),
                      and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
                          in1(g4(_804))))
```

```
Applying Rule 4 to and(in0(g4(_804)),
                       or(neg(in0(g4(_804))),neg(in1(g4(_804)))))
Applying Rule 1A to in0(g4(_804))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
         neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
         neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
         or(neg(x(_804)),neg(y(_804)))
Value of and(x(_804),or(neg(x(_804)),neg(y(_804)))):
         and(x(_804),or(neg(x(_804)),neg(y(_804))))
Applying Rule 4 to and(or(neg(in0(g4(_804))),neg(in1(g4(_804)))),
                       in1(g4(_804)))
Applying Rule 5 to or(neg(in0(g4(_804))),neg(in1(g4(_804))))
Applying Rule 3 to neg(in0(g4(_804)))
Applying Rule 1A to in0(g4(_804))
Value of neg(x(_804)):
         neg(x(_804))
Applying Rule 3 to neg(in1(g4(_804)))
Applying Rule 1A to in1(g4(_804))
Value of neg(y(_804)):
         neg(y(_804))
Value of or(neg(x(_804)),neg(y(_804))):
         or(neg(x(_804)),neg(y(_804)))
Applying Rule 1A to in1(g4(_804))
Value of and(or(neg(x(_804)),neg(y(_804))),y(_804)):
         and(or(neg(x(_804)),neg(y(_804))),y(_804))
Value of or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
            and(or(neg(x(_804)),neg(y(_804))),y(_804))):
         or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
            and(or(neg(x(_804)),neg(y(_804))),y(_804)))
Value of neg(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
                and(or(neg(x(_804)),neg(y(_804))),y(_804)))):
         and(or(neg(x(_804)),and(x(_804),y(_804))),
             or(and(x(_804),y(_804)),neg(y(_804))))
Applying Rule 3 to neg(in1(g5(_804)))
Applying Rule 1A to in1(g5(_804))
Value of neg(cin(_804)):
         neg(cin(_804))
Value of or(and(or(neg(x(_804)),and(x(_804),y(_804))),
```

```
                or(and(x(_804),y(_804)),neg(y(_804)))),
             neg(cin(_804))):
          or(and(or(neg(x(_804)),and(x(_804),y(_804))),
                or(and(x(_804),y(_804)),neg(y(_804)))),
             neg(cin(_804)))
Applying Rule 1A to in1(g5(_804))
Value of and(or(and(or(neg(x(_804)),and(x(_804),y(_804))),
                    or(and(x(_804),y(_804)),neg(y(_804)))),
                neg(cin(_804))),
             cin(_804)):
          and(or(and(or(neg(x(_804)),and(x(_804),y(_804))),
                     or(and(x(_804),y(_804)),neg(y(_804)))),
                neg(cin(_804))),
             cin(_804))
Value of or(and(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
                   and(or(neg(x(_804)),neg(y(_804))),y(_804))),
                or(and(or(neg(x(_804)),and(x(_804),y(_804))),
                       or(and(x(_804),y(_804)),neg(y(_804)))),
                   neg(cin(_804)))),
             and(or(and(or(neg(x(_804)),and(x(_804),y(_804))),
                        or(and(x(_804),y(_804)),neg(y(_804)))),
                    neg(cin(_804))),
                cin(_804))):
          or(and(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
                    and(or(neg(x(_804)),neg(y(_804))),y(_804))),
                 or(and(or(neg(x(_804)),and(x(_804),y(_804))),
                        or(and(x(_804),y(_804)),neg(y(_804)))),
                    neg(cin(_804)))),
             and(or(and(or(neg(x(_804)),and(x(_804),y(_804))),
                        or(and(x(_804),y(_804)),neg(y(_804)))),
                    neg(cin(_804))),
                 cin(_804)))
Does or(and(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
               and(or(neg(x(_804)),neg(y(_804))),y(_804))),
            or(and(or(neg(x(_804)),and(x(_804),y(_804))),
                   or(and(x(_804),y(_804)),neg(y(_804)))),
               neg(cin(_804)))),
         and(or(and(or(neg(x(_804)),and(x(_804),y(_804))),
                    or(and(x(_804),y(_804)),neg(y(_804)))),
                neg(cin(_804))),
             cin(_804))) =
     xor(xor(x(_804),y(_804)),cin(_804))
or(and(or(and(x(_804),or(neg(x(_804)),neg(y(_804)))),
          and(or(neg(x(_804)),neg(y(_804))),y(_804))),
       or(and(or(neg(x(_804)),and(x(_804),y(_804))),
```

```
            or(and(x(_804),y(_804)),neg(y(_804)))),
         neg(cin(_804)))),
   and(or(and(or(neg(x(_804)),and(x(_804),y(_804))),
            or(and(x(_804),y(_804)),neg(y(_804)))),
         neg(cin(_804))),
      cin(_804))) =
xor(xor(x(_804),y(_804)),cin(_804))
By Boolean Expansion
<<< Success! Behavior of faddxor meets its specification.
yes
| ?- halt.
```

```
/*************************************************************/
/*  This file loads the appropriate files to verify a        */
/*  a fulladder with PROLOG-1.  This is invoked by typing    */
/*     ['a:faddload'].                                       */
/*************************************************************/

?- ['a:ops','a:eval','a:derbeh','a:derstate','a:xor','a:faddxor'].
?- ['a:boole2','a:eqbeh','a:verify'].

     Sample run for a full-adder circuit on PROLOG-1:

B:\>prolog

+-------------------------------------------+
| MS-DOS Prolog-1              Version 2.2  |
| Copyright 1983      Serial number: 0001213 |
| Expert Systems Ltd.                       |
| Oxford U.K.                               |
+-------------------------------------------+

?- ['a:faddload'].
a:ops consulted.
a:eval consulted.
a:derbeh consulted.
a:derstate consulted.
a:xor consulted.
a:faddxor consulted.
a:boole2 consulted.
a:eqbeh consulted.
a:verify consulted.
a:faddload consulted.
?- verify(faddxor).
>>> Attempting to verify faddxor>>>
>>>nand2 primitive (needs no verification)>>>
>>>nand2 previously verified >>>
>>>nand2 previously verified >>>
>>> Attempting to verify xor>>>
>>>nand2 previously verified >>>
>>>nand2 previously verified >>>
>>>nand2 previously verified >>>
>>>nand2 previously verified >>>
Applying Rule 1B to out(_109)
Applying Rule 2A to out(g4(_109))
nand2's output equation:
   out(g4(_109)) := or(neg(in0(g4(_109))),neg(in1(g4(_109))))
```

```
Applying Rule 5 to or(neg(in0(g4(_109))),neg(in1(g4(_109))))
Applying Rule 3 to neg(in0(g4(_109)))
Applying Rule 1B to in0(g4(_109))
Applying Rule 2A to out(g2(_109))
nand2's output equation:
   out(g2(_109)) := or(neg(in0(g2(_109))),neg(in1(g2(_109))))
Applying Rule 5 to or(neg(in0(g2(_109))),neg(in1(g2(_109))))
Applying Rule 3 to neg(in0(g2(_109)))
Applying Rule 1A to in0(g2(_109))
Value of neg(in0(_109)):
          neg(in0(_109))
Applying Rule 3 to neg(in1(g2(_109)))
Applying Rule 1B to in1(g2(_109))
Applying Rule 2A to out(g1(_109))
nand2's output equation:
   out(g1(_109)) := or(neg(in0(g1(_109))),neg(in1(g1(_109))))
Applying Rule 5 to or(neg(in0(g1(_109))),neg(in1(g1(_109))))
Applying Rule 3 to neg(in0(g1(_109)))
Applying Rule 1A to in0(g1(_109))
Value of neg(in0(_109)):
          neg(in0(_109))
Applying Rule 3 to neg(in1(g1(_109)))
Applying Rule 1A to in1(g1(_109))
Value of neg(in1(_109)):
          neg(in1(_109))
Value of or(neg(in0(_109)),neg(in1(_109))):
          or(neg(in0(_109)),neg(in1(_109)))
Value of neg(or(neg(in0(_109)),neg(in1(_109)))):
          and(in0(_109),in1(_109))
Value of or(neg(in0(_109)),and(in0(_109),in1(_109))):
          or(neg(in0(_109)),and(in0(_109),in1(_109)))
Value of neg(or(neg(in0(_109)),and(in0(_109),in1(_109)))):
          and(in0(_109),or(neg(in0(_109)),neg(in1(_109))))
Applying Rule 3 to neg(in1(g4(_109)))
Applying Rule 1B to in1(g4(_109))
Applying Rule 2A to out(g3(_109))
nand2's output equation:
   out(g3(_109)) := or(neg(in0(g3(_109))),neg(in1(g3(_109))))
Applying Rule 5 to or(neg(in0(g3(_109))),neg(in1(g3(_109))))
Applying Rule 3 to neg(in0(g3(_109)))
Applying Rule 1B to in0(g3(_109))
Applying Rule 2A to out(g1(_109))
nand2's output equation:
   out(g1(_109)) := or(neg(in0(g1(_109))),neg(in1(g1(_109))))
Applying Rule 5 to or(neg(in0(g1(_109))),neg(in1(g1(_109))))
```

```
Applying Rule 3 to neg(in0(g1(_109)))
Applying Rule 1A to in0(g1(_109))
Value of neg(in0(_109)):
          neg(in0(_109))
Applying Rule 3 to neg(in1(g1(_109)))
Applying Rule 1A to in1(g1(_109))
Value of neg(in1(_109)):
          neg(in1(_109))
Value of or(neg(in0(_109)),neg(in1(_109))):
          or(neg(in0(_109)),neg(in1(_109)))
Value of neg(or(neg(in0(_109)),neg(in1(_109)))):
          and(in0(_109),in1(_109))
Applying Rule 3 to neg(in1(g3(_109)))
Applying Rule 1A to in1(g3(_109))
Value of neg(in1(_109)):
          neg(in1(_109))
Value of or(and(in0(_109),in1(_109)),neg(in1(_109))):
          or(and(in0(_109),in1(_109)),neg(in1(_109)))
Value of neg(or(and(in0(_109),in1(_109)),neg(in1(_109)))):
          and(or(neg(in0(_109)),neg(in1(_109))),in1(_109))
Value of or(and(in0(_109),or(neg(in0(_109)),neg(in1(_109)))),
             and(or(neg(in0(_109)),neg(in1(_109))),in1(_109))):
          or(and(in0(_109),or(neg(in0(_109)),neg(in1(_109)))),
             and(or(neg(in0(_109)),neg(in1(_109))),in1(_109)))
Does or(and(in0(_109),or(neg(in0(_109)),neg(in1(_109)))),
        and(or(neg(in0(_109)),neg(in1(_109))),in1(_109))) =
     or(and(neg(in0(_109)),in1(_109)),and(in0(_109),neg(in1(_109))))
   or(and(in0(_109),or(neg(in0(_109)),neg(in1(_109)))),
      and(or(neg(in0(_109)),neg(in1(_109))),in1(_109))) =
   or(and(neg(in0(_109)),in1(_109)),and(in0(_109),neg(in1(_109))))
By Boolean Expansion
<<< Success! Behavior of xor meets its specification.
>>>xor previously verified >>>
Applying Rule 1B to outcarry(_115)
Applying Rule 2A to out(g3(_115))
nand2's output equation:
   out(g3(_115)) := or(neg(in0(g3(_115))),neg(in1(g3(_115))))
Applying Rule 5 to or(neg(in0(g3(_115))),neg(in1(g3(_115))))
Applying Rule 3 to neg(in0(g3(_115)))
Applying Rule 1B to in0(g3(_115))
Applying Rule 2A to out(g1(_115))
nand2's output equation:
   out(g1(_115)) := or(neg(in0(g1(_115))),neg(in1(g1(_115))))
Applying Rule 5 to or(neg(in0(g1(_115))),neg(in1(g1(_115))))
Applying Rule 3 to neg(in0(g1(_115)))
```

```
Applying Rule 1A to in0(g1(_115))
Value of neg(x(_115)):
          neg(x(_115))
Applying Rule 3 to neg(in1(g1(_115)))
Applying Rule 1A to in1(g1(_115))
Value of neg(y(_115)):
          neg(y(_115))
Value of or(neg(x(_115)),neg(y(_115))):
          or(neg(x(_115)),neg(y(_115)))
Value of neg(or(neg(x(_115)),neg(y(_115)))):
          and(x(_115),y(_115))
Applying Rule 3 to neg(in1(g3(_115)))
Applying Rule 1B to in1(g3(_115))
Applying Rule 2A to out(g2(_115))
nand2's output equation:
   out(g2(_115)) := or(neg(in0(g2(_115))),neg(in1(g2(_115))))
Applying Rule 5 to or(neg(in0(g2(_115))),neg(in1(g2(_115))))
Applying Rule 3 to neg(in0(g2(_115)))
Applying Rule 1A to in0(g2(_115))
Value of neg(cin(_115)):
          neg(cin(_115))
Applying Rule 3 to neg(in1(g2(_115)))
Applying Rule 1B to in1(g2(_115))
Applying Rule 2B to out(g4(_115))
xor's derived behavior:
   out(g4(_115)) := or(and(in0(g4(_115)),
                           or(neg(in0(g4(_115))),neg(in1(g4(_115))))),
                       and(or(neg(in0(g4(_115))),neg(in1(g4(_115)))),
                           in1(g4(_115))))
Applying Rule 5 to or(and(in0(g4(_115)),
                          or(neg(in0(g4(_115))),neg(in1(g4(_115))))),
                      and(or(neg(in0(g4(_115))),neg(in1(g4(_115)))),
                          in1(g4(_115))))
Applying Rule 4 to and(in0(g4(_115)),
                       or(neg(in0(g4(_115))),neg(in1(g4(_115)))))
Applying Rule 1A to in0(g4(_115))
Applying Rule 5 to or(neg(in0(g4(_115))),neg(in1(g4(_115))))
Applying Rule 3 to neg(in0(g4(_115)))
Applying Rule 1A to in0(g4(_115))
Value of neg(x(_115)):
          neg(x(_115))
Applying Rule 3 to neg(in1(g4(_115)))
Applying Rule 1A to in1(g4(_115))
Value of neg(y(_115)):
          neg(y(_115))
```

```
Value of or(neg(x(_115)),neg(y(_115))):
         or(neg(x(_115)),neg(y(_115)))
Value of and(x(_115),or(neg(x(_115)),neg(y(_115)))):
         and(x(_115),or(neg(x(_115)),neg(y(_115))))
Applying Rule 4 to and(or(neg(in0(g4(_115))),neg(in1(g4(_115)))),
                       in1(g4(_115)))
Applying Rule 5 to or(neg(in0(g4(_115))),neg(in1(g4(_115))))
Applying Rule 3 to neg(in0(g4(_115)))
Applying Rule 1A to in0(g4(_115))
Value of neg(x(_115)):
         neg(x(_115))
Applying Rule 3 to neg(in1(g4(_115)))
Applying Rule 1A to in1(g4(_115))
Value of neg(y(_115)):
         neg(y(_115))
Value of or(neg(x(_115)),neg(y(_115))):
         or(neg(x(_115)),neg(y(_115)))
Applying Rule 1A to in1(g4(_115))
Value of and(or(neg(x(_115)),neg(y(_115))),y(_115)):
         and(or(neg(x(_115)),neg(y(_115))),y(_115))
Value of or(and(x(_115),or(neg(x(_115)),neg(y(_115)))),
           and(or(neg(x(_115)),neg(y(_115))),y(_115))):
         or(and(x(_115),or(neg(x(_115)),neg(y(_115)))),
           and(or(neg(x(_115)),neg(y(_115))),y(_115)))
Value of neg(or(and(x(_115),or(neg(x(_115)),neg(y(_115)))),
                and(or(neg(x(_115)),neg(y(_115))),y(_115)))):
         and(or(neg(x(_115)),and(x(_115),y(_115))),
             or(and(x(_115),y(_115)),neg(y(_115))))
Value of or(neg(cin(_115)),and(or(neg(x(_115)),and(x(_115),y(_115))),
                               or(and(x(_115),y(_115)),neg(y(_115))))):
         or(neg(cin(_115)),and(or(neg(x(_115)),and(x(_115),y(_115))),
                               or(and(x(_115),y(_115)),neg(y(_115)))))
Value of neg(or(neg(cin(_115)),
                and(or(neg(x(_115)),and(x(_115),y(_115))),
                    or(and(x(_115),y(_115)),neg(y(_115))))))):
         and(cin(_115),or(and(x(_115),or(neg(x(_115)),neg(y(_115)))),
                          and(or(neg(x(_115)),neg(y(_115))),y(_115))))
Value of or(and(x(_115),y(_115)),
           and(cin(_115),or(and(x(_115),or(neg(x(_115)),neg(y(_115)))),
                            and(or(neg(x(_115)),neg(y(_115))),y(_115))))):
         or(and(x(_115),y(_115)),
           and(cin(_115),or(and(x(_115),or(neg(x(_115)),neg(y(_115)))),
                            and(or(neg(x(_115)),neg(y(_115))),y(_115)))))
Does or(and(x(_115),y(_115)),
        and(cin(_115),or(and(x(_115),or(neg(x(_115)),neg(y(_115)))),
```

```
                        and(or(neg(x(_115)),neg(y(_115))),y(_115))))) =
    or(and(x(_115),y(_115)),and(cin(_115),xor(x(_115),y(_115))))
 or(and(x(_115),y(_115)),
    and(cin(_115),or(and(x(_115),or(neg(x(_115)),neg(y(_115)))),
                     and(or(neg(x(_115)),neg(y(_115))),y(_115))))) =
 or(and(x(_115),y(_115)),and(cin(_115),xor(x(_115),y(_115))))
By Boolean Expansion
Applying Rule 1B to outsum(_114)
Applying Rule 2B to out(g5(_114))
xor's derived behavior:
   out(g5(_114)) := or(and(in0(g5(_114)),
                           or(neg(in0(g5(_114))),neg(in1(g5(_114))))),
                       and(or(neg(in0(g5(_114))),neg(in1(g5(_114)))),
                           in1(g5(_114))))
Applying Rule 5 to or(and(in0(g5(_114)),
                          or(neg(in0(g5(_114))),neg(in1(g5(_114))))),
                      and(or(neg(in0(g5(_114))),neg(in1(g5(_114)))),
                          in1(g5(_114))))
Applying Rule 4 to and(in0(g5(_114)),
                       or(neg(in0(g5(_114))),neg(in1(g5(_114)))))
Applying Rule 1B to in0(g5(_114))
Applying Rule 2B to out(g4(_114))
xor's derived behavior:
   out(g4(_114)) := or(and(in0(g4(_114)),
                           or(neg(in0(g4(_114))),neg(in1(g4(_114))))),
                       and(or(neg(in0(g4(_114))),neg(in1(g4(_114)))),
                           in1(g4(_114))))
Applying Rule 5 to or(and(in0(g4(_114)),
                          or(neg(in0(g4(_114))),neg(in1(g4(_114))))),
                      and(or(neg(in0(g4(_114))),neg(in1(g4(_114)))),
                          in1(g4(_114))))
Applying Rule 4 to and(in0(g4(_114)),
                       or(neg(in0(g4(_114))),neg(in1(g4(_114)))))
Applying Rule 1A to in0(g4(_114))
Applying Rule 5 to or(neg(in0(g4(_114))),neg(in1(g4(_114))))
Applying Rule 3 to neg(in0(g4(_114)))
Applying Rule 1A to in0(g4(_114))
Value of neg(x(_114)):
         neg(x(_114))
Applying Rule 3 to neg(in1(g4(_114)))
Applying Rule 1A to in1(g4(_114))
Value of neg(y(_114)):
         neg(y(_114))
Value of or(neg(x(_114)),neg(y(_114))):
         or(neg(x(_114)),neg(y(_114)))
```

```
Value of and(x(_114),or(neg(x(_114)),neg(y(_114)))):
          and(x(_114),or(neg(x(_114)),neg(y(_114))))
Applying Rule 4 to and(or(neg(in0(g4(_114))),neg(in1(g4(_114)))),
                       in1(g4(_114)))
Applying Rule 5 to or(neg(in0(g4(_114))),neg(in1(g4(_114))))
Applying Rule 3 to neg(in0(g4(_114)))
Applying Rule 1A to in0(g4(_114))
Value of neg(x(_114)):
          neg(x(_114))
Applying Rule 3 to neg(in1(g4(_114)))
Applying Rule 1A to in1(g4(_114))
Value of neg(y(_114)):
          neg(y(_114))
Value of or(neg(x(_114)),neg(y(_114))):
          or(neg(x(_114)),neg(y(_114)))
Applying Rule 1A to in1(g4(_114))
Value of and(or(neg(x(_114)),neg(y(_114))),y(_114)):
          and(or(neg(x(_114)),neg(y(_114))),y(_114))
Value of or(and(x(_114),or(neg(x(_114)),neg(y(_114)))),
            and(or(neg(x(_114)),neg(y(_114))),y(_114))):
          or(and(x(_114),or(neg(x(_114)),neg(y(_114)))),
            and(or(neg(x(_114)),neg(y(_114))),y(_114)))
Applying Rule 5 to or(neg(in0(g5(_114))),neg(in1(g5(_114))))
Applying Rule 3 to neg(in0(g5(_114)))
Applying Rule 1B to in0(g5(_114))
Applying Rule 2B to out(g4(_114))
xor's derived behavior:
   out(g4(_114)) := or(and(in0(g4(_114)),
                           or(neg(in0(g4(_114))),neg(in1(g4(_114))))),
                       and(or(neg(in0(g4(_114))),neg(in1(g4(_114)))),
                           in1(g4(_114))))
Applying Rule 5 to or(and(in0(g4(_114)),
                          or(neg(in0(g4(_114))),neg(in1(g4(_114))))),
                      and(or(neg(in0(g4(_114))),neg(in1(g4(_114)))),
                          in1(g4(_114))))
Applying Rule 4 to and(in0(g4(_114)),
                       or(neg(in0(g4(_114))),neg(in1(g4(_114)))))
Applying Rule 1A to in0(g4(_114))
Applying Rule 5 to or(neg(in0(g4(_114))),neg(in1(g4(_114))))
Applying Rule 3 to neg(in0(g4(_114)))
Applying Rule 1A to in0(g4(_114))
Value of neg(x(_114)):
          neg(x(_114))
Applying Rule 3 to neg(in1(g4(_114)))
Applying Rule 1A to in1(g4(_114))
```

```
Value of neg(y(_114)):
         neg(y(_114))
Value of or(neg(x(_114)),neg(y(_114))):
         or(neg(x(_114)),neg(y(_114)))
Value of and(x(_114),or(neg(x(_114)),neg(y(_114)))):
         and(x(_114),or(neg(x(_114)),neg(y(_114))))
Applying Rule 4 to and(or(neg(in0(g4(_114))),neg(in1(g4(_114)))),
                       in1(g4(_114)))
Applying Rule 5 to or(neg(in0(g4(_114))),neg(in1(g4(_114))))
Applying Rule 3 to neg(in0(g4(_114)))
Applying Rule 1A to in0(g4(_114))
Value of neg(x(_114)):
         neg(x(_114))
Applying Rule 3 to neg(in1(g4(_114)))
Applying Rule 1A to in1(g4(_114))
Value of neg(y(_114)):
         neg(y(_114))
Value of or(neg(x(_114)),neg(y(_114))):
         or(neg(x(_114)),neg(y(_114)))
Applying Rule 1A to in1(g4(_114))
Value of and(or(neg(x(_114)),neg(y(_114))),y(_114)):
         and(or(neg(x(_114)),neg(y(_114))),y(_114))
Value of or(and(x(_114),or(neg(x(_114)),neg(y(_114)))),
            and(or(neg(x(_114)),neg(y(_114))),y(_114))):
         or(and(x(_114),or(neg(x(_114)),neg(y(_114)))),
            and(or(neg(x(_114)),neg(y(_114))),y(_114)))
Value of neg(or(and(x(_114),or(neg(x(_114)),neg(y(_114)))),
                and(or(neg(x(_114)),neg(y(_114))),y(_114)))):
         and(or(neg(x(_114)),and(x(_114),y(_114))),
             or(and(x(_114),y(_114)),neg(y(_114))))
Applying Rule 3 to neg(in1(g5(_114)))
Applying Rule 1A to in1(g5(_114))
Value of neg(cin(_114)):
         neg(cin(_114))
Value of or(and(or(neg(x(_114)),and(x(_114),y(_114))),
                or(and(x(_114),y(_114)),neg(y(_114)))),
            neg(cin(_114))):
         or(and(or(neg(x(_114)),and(x(_114),y(_114))),
                or(and(x(_114),y(_114)),neg(y(_114)))),
            neg(cin(_114)))
Value of and(or(and(x(_114),or(neg(x(_114)),neg(y(_114)))),
                and(or(neg(x(_114)),neg(y(_114))),
                    y(_114))),
             or(and(or(neg(x(_114)),and(x(_114),y(_114))),
                    or(and(x(_114),y(_114)),
```

```
                    neg(y(_114)))),
                neg(cin(_114)))):
        and(or(and(x(_114),or(neg(x(_114)),neg(y(_114)))),
               and(or(neg(x(_114)),neg(y(_114))),
                   y(_114))),
            or(and(or(neg(x(_114)),and(x(_114),y(_114))),
                   or(and(x(_114),y(_114)),
                      neg(y(_114)))),
               neg(cin(_114))))
Applying Rule 4 to and(or(neg(in0(g5(_114))),neg(in1(g5(_114)))),
                       in1(g5(_114)))
Applying Rule 5 to or(neg(in0(g5(_114))),neg(in1(g5(_114))))
Applying Rule 3 to neg(in0(g5(_114)))
Applying Rule 1B to in0(g5(_114))
Applying Rule 2B to out(g4(_114))
xor's derived behavior:
   out(g4(_114)) := or(and(in0(g4(_114)),
                           or(neg(in0(g4(_114))),neg(in1(g4(_114))))),
                       and(or(neg(in0(g4(_114))),neg(in1(g4(_114)))),
                           in1(g4(_114))))
Applying Rule 5 to or(and(in0(g4(_114)),
                          or(neg(in0(g4(_114))),neg(in1(g4(_114))))),
                      and(or(neg(in0(g4(_114))),neg(in1(g4(_114)))),
                          in1(g4(_114))))
Applying Rule 4 to and(in0(g4(_114)),
                       or(neg(in0(g4(_114))),neg(in1(g4(_114)))))
Applying Rule 1A to in0(g4(_114))
Applying Rule 5 to or(neg(in0(g4(_114))),neg(in1(g4(_114))))
Applying Rule 3 to neg(in0(g4(_114)))
Applying Rule 1A to in0(g4(_114))
Value of neg(x(_114)):
         neg(x(_114))
Applying Rule 3 to neg(in1(g4(_114)))
Applying Rule 1A to in1(g4(_114))
Value of neg(y(_114)):
         neg(y(_114))
Value of or(neg(x(_114)),neg(y(_114))):
         or(neg(x(_114)),neg(y(_114)))
Value of and(x(_114),or(neg(x(_114)),neg(y(_114)))):
         and(x(_114),or(neg(x(_114)),neg(y(_114))))
Applying Rule 4 to and(or(neg(in0(g4(_114))),neg(in1(g4(_114)))),
                       in1(g4(_114)))
Applying Rule 5 to or(neg(in0(g4(_114))),neg(in1(g4(_114))))
Applying Rule 3 to neg(in0(g4(_114)))
Applying Rule 1A to in0(g4(_114))
```

```
Value of neg(x(_114)):
          neg(x(_114))
Applying Rule 3 to neg(in1(g4(_114)))
Applying Rule 1A to in1(g4(_114))
Value of neg(y(_114)):
          neg(y(_114))
Value of or(neg(x(_114)),neg(y(_114))):
          or(neg(x(_114)),neg(y(_114)))
Applying Rule 1A to in1(g4(_114))
Value of and(or(neg(x(_114)),neg(y(_114))),y(_114)):
          and(or(neg(x(_114)),neg(y(_114))),y(_114))
Value of or(and(x(_114),or(neg(x(_114)),neg(y(_114)))),
            and(or(neg(x(_114)),neg(y(_114))),y(_114))):
          or(and(x(_114),or(neg(x(_114)),neg(y(_114)))),
            and(or(neg(x(_114)),neg(y(_114))),y(_114)))
Value of neg(or(and(x(_114),or(neg(x(_114)),neg(y(_114)))),
                and(or(neg(x(_114)),neg(y(_114))),y(_114)))):
          and(or(neg(x(_114)),and(x(_114),y(_114))),
              or(and(x(_114),y(_114)),neg(y(_114))))
Applying Rule 3 to neg(in1(g5(_114)))
Applying Rule 1A to in1(g5(_114))
Value of neg(cin(_114)):
          neg(cin(_114))
Value of or(and(or(neg(x(_114)),and(x(_114),y(_114))),
                or(and(x(_114),y(_114)),neg(y(_114)))),
            neg(cin(_114))):
          or(and(or(neg(x(_114)),and(x(_114),y(_114))),
                or(and(x(_114),y(_114)),neg(y(_114)))),
            neg(cin(_114)))
Applying Rule 1A to in1(g5(_114))
Value of and(or(and(or(neg(x(_114)),and(x(_114),y(_114))),
                    or(and(x(_114),y(_114)),neg(y(_114)))),
                neg(cin(_114))),
              cin(_114)):
          and(or(and(or(neg(x(_114)),and(x(_114),y(_114))),
                    or(and(x(_114),y(_114)),neg(y(_114)))),
                neg(cin(_114))),
              cin(_114))
Value of or(and(or(and(x(_114),or(neg(x(_114)),neg(y(_114)))),
                   and(or(neg(x(_114)),neg(y(_114))),y(_114))),
                or(and(or(neg(x(_114)),and(x(_114),y(_114))),
                       or(and(x(_114),y(_114)),neg(y(_114)))),
                   neg(cin(_114)))),
            and(or(and(or(neg(x(_114)),and(x(_114),y(_114))),
                       or(and(x(_114),y(_114)),neg(y(_114)))),
```

```
                neg(cin(_114))),
              cin(_114))):
        or(and(or(and(x(_114),or(neg(x(_114)),neg(y(_114)))),
                  and(or(neg(x(_114)),neg(y(_114))),y(_114))),
              or(and(or(neg(x(_114)),and(x(_114),y(_114))),
                     or(and(x(_114),y(_114)),neg(y(_114)))),
                  neg(cin(_114)))),
           and(or(and(or(neg(x(_114)),and(x(_114),y(_114))),
                     or(and(x(_114),y(_114)),neg(y(_114)))),
                  neg(cin(_114))),
              cin(_114)))
Does or(and(or(and(x(_114),or(neg(x(_114)),neg(y(_114)))),
               and(or(neg(x(_114)),neg(y(_114))),y(_114))),
            or(and(or(neg(x(_114)),and(x(_114),y(_114))),
                   or(and(x(_114),y(_114)),neg(y(_114)))),
               neg(cin(_114)))),
        and(or(and(or(neg(x(_114)),and(x(_114),y(_114))),
                   or(and(x(_114),y(_114)),neg(y(_114)))),
               neg(cin(_114))),
            cin(_114))) =
      xor(xor(x(_114),y(_114)),cin(_114))

Error number: 83                    /*   stack space exceeded Error  */
No error file
Evaluation Aborted
?- halt.
```

Notice that the fulladder was unable to complete on Prolog-1. This is due to insufficient stack space while determining the equivalence of the specified and derived sum output. Quintus Prolog had no such problem and it would appear Prolog-1 has outlived its usefulness for this project.

Bibliography

1. Barrow, Harry G. "VERIFY: A Program for Proving Correctness of Digital Hardware Designs," *Artificial Intelligence*, *24*:437–491 (December 1984).
2. Bratko, Ivan. *Prolog Programming for Artificial Intelligence*. Addison-Wesley Publishing Company, Inc., 1986.
3. Brezocnik, Z., B. Horvat and M. Gerkes. "Tool for System Design Verification." In *Proceedings of the CompEuro 88 - System Design: Concepts, Methods and Tools*, pages 100–107, Washington D.C.: IEEE Computer Society Press, 1988.
4. Clocksin, W.F. "Logic programming and digital circuit analysis," *Journal of Logic Programming*, *4*:59–82 (January 1987).
5. Clocksin, W.F. and C. S. Mellish. *Programming in Prolog*. Springer-Verlag, 1981.
6. Dukes, CPT Michael A. "Formal Verification Using VHDL." PhD Prospectus, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, March 1990.
7. Expert Systems International. *8086 PROLOG-1 Reference Manual*, December 1983.
8. Gordon, Mike. *Hardware Verification by Formal Proof*. Technical Report 74, University of Cambridge Computer Laboratory, August 1985.
9. Gordon, Mike. "Why higher-order logic is a good formalism for specifying and verifying hardware." In Milne, G.J. and P.A. Subrahmanyam., editors, *Formal Aspects of VLSI Design, Proceedings of the 1985 Edinburgh Workshop on VLSI*, pages 153–177, Amsterdam, Netherlands: North-Holland, 1986.
10. Grabowiecki, T., A. Pawlak and W. Sakowski. "A University Framework for Correct by Construction IC Design," *Microprocessing and Microprogramming*, *23*:37–43 (1988).
11. Hwang, Seung H. and A.R. Newton. "An Efficient Design Correctness Checker of Finite State Machines." In *Proceedings of the Fifth IEEE International Conference on Computer-Aided Design*, pages 410–413, Washington D.C.: IEEE Computer Society Press, 1987.
12. Maruyama, Fumihiro and Masahiro Fujita. "A Verification Technique for Hardware Designs," *IEEE Computer*, *18*:22–40 (February 1985).
13. Robinson, A.J. "A Machine-oriented Logic Based on the Resoultion Principle," *Journal of the Association of Computer Machinery*, *12*:23–41 (1965).